

DSP System Toolbox™

User's Guide

R2012a

**MATLAB®
& SIMULINK®**

How to Contact MathWorks



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



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



508-647-7000 (Phone)



508-647-7001 (Fax)



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

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

DSP System Toolbox™ User's Guide

© COPYRIGHT 2011–2012 by MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

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

Patents

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

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)

Input, Output, and Display

1

Discrete-Time Signals	1-2
Time and Frequency Terminology	1-2
Recommended Settings for Discrete-Time Simulations ...	1-4
Other Settings for Discrete-Time Simulations	1-6
Continuous-Time Signals	1-11
Continuous-Time Source Blocks	1-11
Continuous-Time Nonsource Blocks	1-11
Create Sample-Based Signals	1-13
Create a Sample-Based Signal Using the Constant Block	1-13
Create a Sample-Based Signal Using the Signal from Workspace Block	1-16
Create Frame-Based Signals	1-19
Create a Frame-Based Signal Using the Sine Wave Block	1-19
Create a Frame-Based Signal Using the Signal from Workspace Block	1-22
Create Multichannel Sample-Based Signals	1-26
Multichannel Sample-Based Signals	1-26
Create a Multichannel Sample-Based Signal by Combining Single-Channel Sample-Based Signals	1-26
Create a Multichannel Sample-Based Signal by Combining Multichannel Sample-Based Signals	1-29
Create Multichannel Frame-Based Signals	1-32
Multichannel Frame-Based Signals	1-32
Create a Multichannel Frame-Based Signal Using the Concatenate Block	1-33

Deconstruct Multichannel Sample-Based Signals	1-36
Split Multichannel Sample-Based Signals into Individual Signals	1-36
Split Multichannel Sample-Based Signals into Several Multichannel Signals	1-39
Deconstruct Multichannel Frame-Based Signals	1-43
Split Multichannel Frame-Based Signals into Individual Signals	1-43
Reorder Channels in Multichannel Frame-Based Signals	1-48
Import and Export Sample-Based Signals	1-52
Import Sample-Based Vector Signals	1-52
Import Sample-Based Matrix Signals	1-55
Export Sample-Based Signals	1-59
Import and Export Frame-Based Signals	1-64
Import Frame-Based Signals	1-64
Export Frame-Based Signals	1-67
Display Time-Domain Data	1-73
Configure the Time Scope	1-75
Use the Simulation Controls	1-80
Modify the Time Scope Display	1-82
Inspect Your Data (Scaling the Axes and Zooming)	1-84
Manage Multiple Time Scopes	1-86
Display Frequency-Domain Data	1-90

Data and Signal Management

2

Sample- and Frame-Based Concepts	2-2
Sample- and Frame-Based Signals	2-2
Model Sample- and Frame-Based Signals in MATLAB and Simulink	2-3
What Is Sample-Based Processing?	2-4

What Is Frame-Based Processing?	2-5
Inspect Sample Rates and Frame Rates in Simulink ..	2-8
Sample Rate and Frame Rate Concepts	2-8
Inspect Sample-Based Signals Using the Probe Block	2-10
Inspect Frame-Based Signals Using the Probe Block	2-11
Inspect Sample-Based Signals Using Color Coding	2-13
Inspect Frame-Based Signals Using Color Coding	2-15
Convert Sample and Frame Rates in Simulink	2-17
Rate Conversion Blocks	2-17
Rate Conversion by Frame-Rate Adjustment	2-18
Rate Conversion by Frame-Size Adjustment	2-21
Avoid Unintended Rate Conversion	2-24
Frame Rebuffering Blocks	2-30
Buffer Signals by Preserving the Sample Period	2-33
Buffer Signals by Altering the Sample Period	2-36
Convert Frame Status	2-39
Frame Status	2-39
Buffer Sample-Based Signals into Frame-Based Signals ..	2-39
Buffer Sample-Based Signals into Frame-Based Signals with Overlap	2-42
Buffer Frame-Based Signals into Other Frame-Based Signals	2-47
Buffer Delay and Initial Conditions	2-50
Unbuffer Frame-Based Signals into Sample-Based Signals	2-51
Delay and Latency	2-55
Computational Delay	2-55
Algorithmic Delay	2-57
Zero Algorithmic Delay	2-57
Basic Algorithmic Delay	2-60
Excess Algorithmic Delay (Tasking Latency)	2-63
Predict Tasking Latency	2-65

Filter Analysis, Design, and Implementation

3

Design a Filter in Fdesign — Process Overview	3-2
Process Flow Diagram and Filter Design Methodology ...	3-2
Design a Filter in the Filterbuilder GUI	3-11
The Graphical Interface to Fdesign	3-11
Use FDATool with DSP System Toolbox Software	3-16
Design Advanced Filters in FDATool	3-16
Access the Quantization Features of FDATool	3-20
Quantize Filters in FDATool	3-22
Analyze Filters in MATLAB with a Noise-Based Method ..	3-30
Scale Second-Order Section Filters	3-37
Reorder the Sections of Second-Order Section Filters	3-42
View SOS Filter Sections	3-48
Import and Export Quantized Filters	3-53
Generate MATLAB Code	3-58
Import XILINX Coefficient (.COE) Files	3-59
Transform Filters Using FDATool	3-59
Design Multirate Filters in FDATool	3-69
Realize Filters as Simulink Subsystem Blocks	3-83
Digital Frequency Transformations	3-87
Details and Methodology	3-87
Frequency Transformations for Real Filters	3-95
Frequency Transformations for Complex Filters	3-109
Digital Filter Design Block	3-122
Overview of the Digital Filter Design Block	3-122
Select a Filter Design Block	3-123
Create a Lowpass Filter in Simulink	3-125
Create a Highpass Filter in Simulink	3-126
Filter High-Frequency Noise in Simulink	3-128
Filter Realization Wizard	3-134
Overview of the Filter Realization Wizard	3-134
Design and Implement a Fixed-Point Filter in Simulink ..	3-134
Set the Filter Structure and Number of Filter Sections ...	3-143
Optimize the Filter Structure	3-144

Digital Filter Block	3-147
Overview of the Digital Filter Block	3-147
Implement a Lowpass Filter in Simulink	3-148
Implement a Highpass Filter in Simulink	3-149
Filter High-Frequency Noise in Simulink	3-150
Specify Static Filters	3-155
Specify Time-Varying Filters	3-155
Specify the SOS Matrix (Biquadratic Filter Coefficients) ..	3-157
Analog Filter Design Block	3-159
Fixed-Point Filter Design	3-161
Overview of Fixed-Point Filters	3-161
Data Types for Filter Functions	3-161
Convert a Filter from Floating Point to Fixed Point in	
MATLAB	3-163
Create an FIR Filter Using Integer Coefficients	3-171
Fixed-Point Filtering in Simulink	3-188

Adaptive Filters

4

Overview of Adaptive Filters and Applications	4-2
Introduction to Adaptive Filtering	4-2
Adaptive Filtering Methodology	4-2
Choosing an Adaptive Filter	4-4
System Identification	4-6
Inverse System Identification	4-6
Noise or Interference Cancellation	4-7
Prediction	4-8
Adaptive Filters in DSP System Toolbox Software	4-10
Overview of Adaptive Filtering in DSP System Toolbox	
Software	4-10
Algorithms	4-10
Using Adaptive Filter Objects	4-13
LMS Adaptive Filters	4-14
LMS Methods for adaptfilt Objects	4-14

System Identification Using adaptfilt.lms	4-16
System Identification Using adaptfilt.nlms	4-19
Noise Cancellation Using adaptfilt.sd	4-22
Noise Cancellation Using adaptfilt.se	4-26
Noise Cancellation Using adaptfilt.ss	4-31
RLS Adaptive Filters	4-36
Comparison of the RLS and LMS Adaptive Filter	
Algorithms	4-36
Inverse System Identification Using adaptfilt.rls	4-37
Enhance a Signal Using LMS and Normalized LMS	
Algorithms	4-42
Create the Signals for Adaptation	4-42
Construct Two Adaptive Filters	4-43
Choose the Step Size	4-44
Set the Adapting Filter Step Size	4-45
Filter with the Adaptive Filters	4-45
Compute the Optimal Solution	4-45
Plot the Results	4-46
Compare the Final Coefficients	4-47
Reset the Filter Before Filtering	4-47
Investigate Convergence Through Learning Curves	4-48
Compute the Learning Curves	4-49
Compute the Theoretical Learning Curves	4-50
Adaptive Filters in Simulink	4-51
Create an Acoustic Environment in Simulink	4-51
Configure an LMS Filter Block for Adaptive Noise	
Cancellation	4-53
Modify Adaptive Filter Parameters During Model	
Simulation	4-58
Adaptive Filtering Demos	4-62
Selected Bibliography	4-64

Multirate and Multistage Filters

5

Multirate Filters	5-2
Why Are Multirate Filters Needed?	5-2
Overview of Multirate Filters	5-2
Multistage Filters	5-6
Why Are Multistage Filters Needed?	5-6
Optimal Multistage Filters in DSP System Toolbox Software	5-6
Example Case for Multirate/Multistage Filters	5-8
Example Overview	5-8
Single-Rate/Single-Stage Equiripple Design	5-8
Reduce Computational Cost Using Multirate/Multistage Design	5-9
Compare the Responses	5-9
Further Performance Comparison	5-10
Filter Banks	5-12
Dyadic Analysis Filter Banks	5-12
Dyadic Synthesis Filter Banks	5-16
Examples of Multirate Filtering in Simulink	5-21

Transforms, Estimation, and Spectral Analysis

6

Transform Time-Domain Data into the Frequency Domain Using the FFT Block	6-2
Transform Frequency-Domain Data into the Time Domain Using the IFFT Block	6-7
Linear and Bit-Reversed Output Order	6-12
FFT and IFFT Blocks Data Order	6-12

Find the Bit-Reversed Order of Your Frequency Indices . .	6-12
Calculate the Channel Latencies Required for Wavelet	
Reconstruction	6-14
Analyze Your Model	6-14
Calculate the Group Delay of Your Filters	6-16
Reconstruct the Filter Bank System	6-18
Equalize the Delay on Each Filter Path	6-18
Update and Run the Model	6-21
References	6-22
Spectral Analysis	6-23
Power Spectrum Estimates	6-24
Create the Block Diagram	6-24
Set the Model Parameters	6-25
View the Power Spectrum Estimates	6-31
Spectrograms	6-34
Modify the Block Diagram	6-34
Set the Model Parameters	6-36
View the Spectrogram of the Speech Signal	6-40

Mathematics

7

Statistics	7-2
Statistics Blocks	7-2
Basic Operations	7-3
Running Operations	7-4
Linear Algebra	7-6
Linear Algebra Blocks	7-6
Linear System Solvers	7-6
Matrix Factorizations	7-8
Matrix Inverses	7-9

Fixed-Point Signal Processing	8-2
Fixed-Point Features	8-2
Benefits of Fixed-Point Hardware	8-2
Benefits of Fixed-Point Design with System Toolboxes Software	8-3
Fixed-Point Concepts and Terminology	8-4
Fixed-Point Data Types	8-4
Scaling	8-5
Precision and Range	8-6
Arithmetic Operations	8-10
Modulo Arithmetic	8-10
Two's Complement	8-11
Addition and Subtraction	8-12
Multiplication	8-13
Casts	8-16
Fixed-Point Support for MATLAB System Objects	8-21
Get Information About Fixed-Point System Objects	8-21
Display Fixed-Point Properties for System Objects	8-25
Set System Object Fixed-Point Properties	8-26
Full Precision for Fixed-Point System Objects	8-27
Specify Fixed-Point Attributes for Blocks	8-28
Fixed-Point Block Parameters	8-28
Specify System-Level Settings	8-31
Inherit via Internal Rule	8-32
Select and Specify Data Types for Fixed-Point Blocks	8-43
Quantizers	8-51
Scalar Quantizers	8-51
Vector Quantizers	8-58
Fixed-Point Filter Design	8-65
Overview of Fixed-Point Filters	8-65
Data Types for Filter Functions	8-65

Convert a Filter from Floating Point to Fixed Point in MATLAB	8-67
Create an FIR Filter Using Integer Coefficients	8-75
Fixed-Point Filtering in Simulink	8-92

Code Generation

9

Understanding Code Generation	9-2
Code Generation with the Simulink® Coder™ Product ...	9-2
Highly Optimized Generated ANSI C Code	9-3
 Code Generation with System Objects	9-4
 Generate Code from MATLAB	9-9
 Generate Code from Simulink	9-10
Open and Run the Model	9-10
Generate Code from the Model	9-11
Build and Run the Generated Code	9-12

Define New System Objects

10

Define Basic System Objects	10-2
 Change Number of Step Method Inputs or Outputs ...	10-4
 Validate Property and Input Values	10-7
 Initialize Properties and Setup One-Time Calculations	10-10

Set Property Values at Construction from Name-Value Pairs	10-13
Reset Algorithm State	10-16
Define Property Attributes	10-18
Hide Inactive Properties	10-21
Limit Property Values to a Finite Set of Strings	10-23
Process Tuned Properties	10-26
Release System Object Resources	10-28
Define Composite System Objects	10-30
Define Finite Source Objects	10-34
Methods Timing	10-36
Setup Method Call Sequence	10-36
Step Method Call Sequence	10-37
Reset Method Call Sequence	10-37
Release Method Call Sequence	10-38

Bibliography

A

Advanced Filters	A-2
Adaptive Filters	A-3
Multirate Filters	A-4
Frequency Transformations	A-5

Fixed-Point Filters **A-6**

Index

Input, Output, and Display

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

- “Discrete-Time Signals” on page 1-2
- “Continuous-Time Signals” on page 1-11
- “Create Sample-Based Signals” on page 1-13
- “Create Frame-Based Signals” on page 1-19
- “Create Multichannel Sample-Based Signals” on page 1-26
- “Create Multichannel Frame-Based Signals” on page 1-32
- “Deconstruct Multichannel Sample-Based Signals” on page 1-36
- “Deconstruct Multichannel Frame-Based Signals” on page 1-43
- “Import and Export Sample-Based Signals” on page 1-52
- “Import and Export Frame-Based Signals” on page 1-64
- “Display Time-Domain Data” on page 1-73
- “Display Frequency-Domain Data” on page 1-90

Discrete-Time Signals

In this section...

“Time and Frequency Terminology” on page 1-2

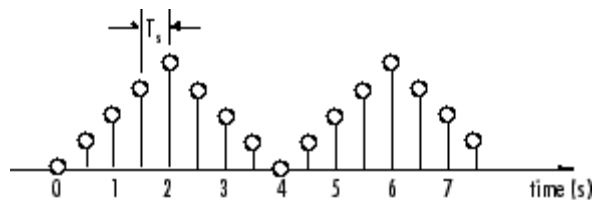
“Recommended Settings for Discrete-Time Simulations” on page 1-4

“Other Settings for Discrete-Time Simulations” on page 1-6

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} T_{so}	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}).
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.

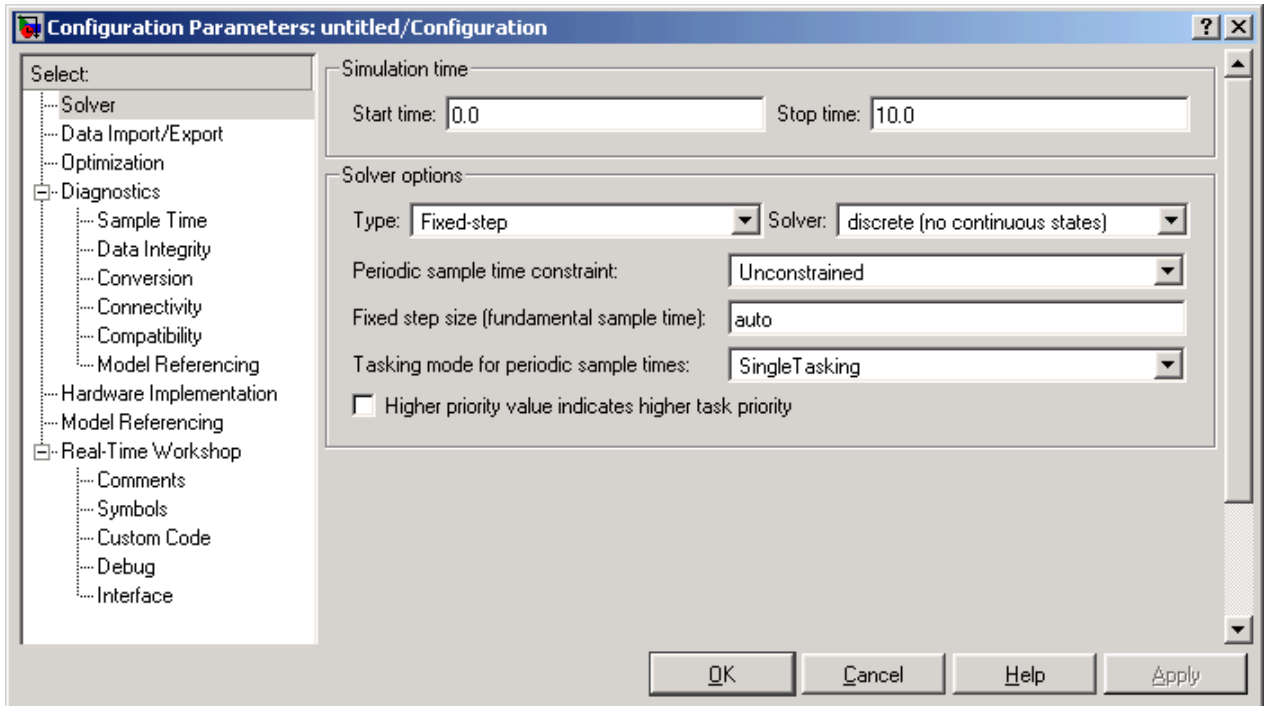
Recommended Settings for Discrete-Time Simulations

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 **Configuration Parameters**. The **Configuration Parameters** dialog box opens.
- 2 In the **Select** pane, click **Solver**.

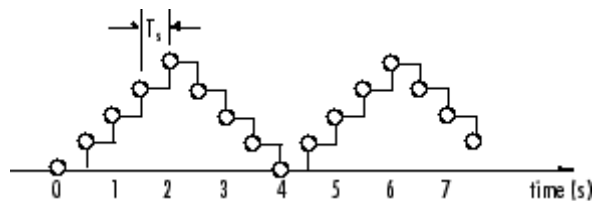
The selections that you make here determine how discrete-time signals are processed in Simulink. The recommended **Solver options** settings for signal processing simulations are

- **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 running the `dspsstartup.m` file. See “Configure the Simulink Environment for Signal Processing Models” in the *DSP System Toolbox Getting Started Guide* for more information.

In **Fixed-step SingleTasking** mode, discrete-time signals differ from the prototype described in “Time and Frequency Terminology” on page 1-2 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 1-7.

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 1-4. 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 2-63 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.

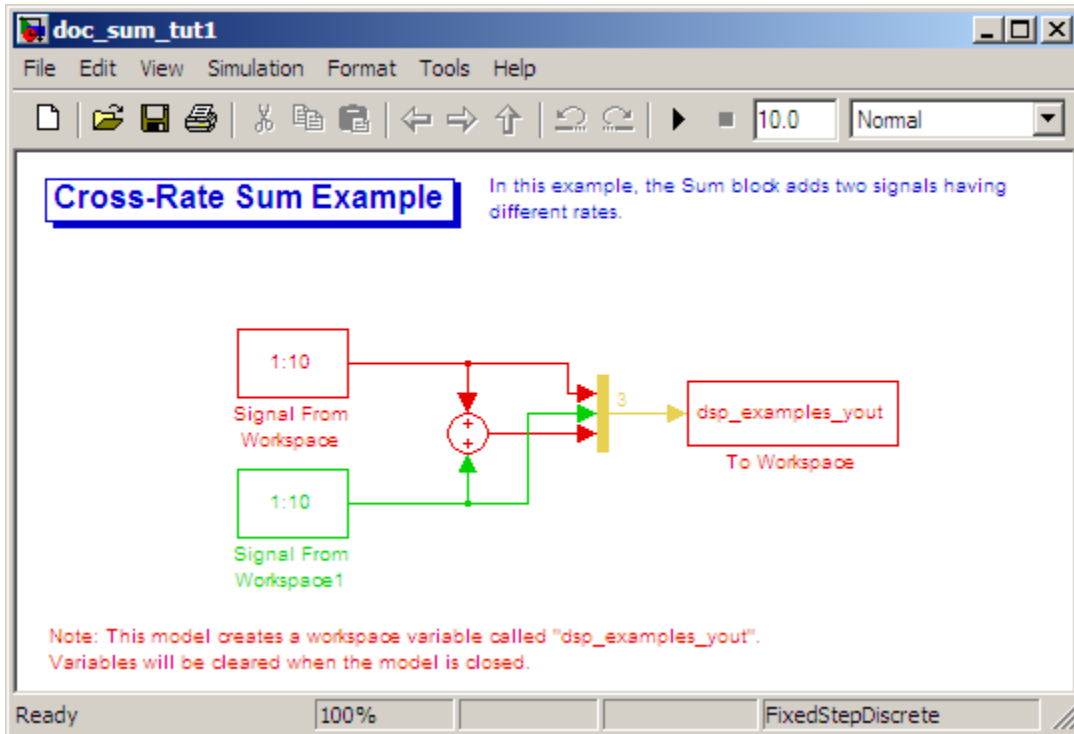
Cross-Rate Operations

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 2-17 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 **Format** menu choose **Sample Time Display > 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 “How to View Sample Time Information” in the Simulink documentation.

- 7 Run the model.

Note Using the `dspstartup` configurations 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...

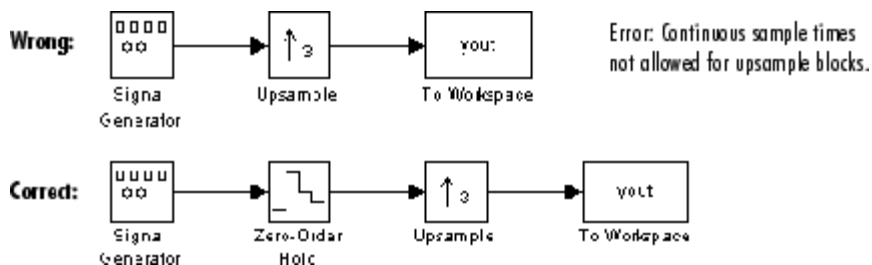
“Continuous-Time Source Blocks” on page 1-11

“Continuous-Time Nonsource Blocks” on page 1-11

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 **Format** menu, you point to **Sample Time Display** 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 Digital Filter, FIR Decimation, FIR Interpolation blocks.

Create Sample-Based Signals

Note Starting in R2010b, many DSP System Toolbox blocks received a new parameter to control whether they perform sample- or frame-based processing. The following content has not been updated to reflect this change. For more information, see the “Frame-Based Processing” section of the Release Notes.

In this section...

“Create a Sample-Based Signal Using the Constant Block” on page 1-13

“Create a Sample-Based Signal Using the Signal from Workspace Block” on page 1-16

Create a Sample-Based Signal Using the Constant Block

A constant sample-based signal has identical successive samples. The Sources library provides the following blocks for creating constant sample-based signals:

- Constant Diagonal Matrix
- Constant
- Identity Matrix

The most versatile of the blocks listed above is the Constant block. This topic discusses how to create a constant sample-based signal using the 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
- **Sampling Mode** = Sample based
- **Sample time** = 1

Based on these parameters, the Constant block outputs a constant, discrete-valued, sample-based 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. See “Linear Algebra” in the MATLAB documentation for a thorough introduction to constructing and indexing matrices.

- 6 Save these parameters and close the dialog box by clicking **OK**.
- 7 From the **Format** menu, point to **Port/ Signal Displays** 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, constant sample-based 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:

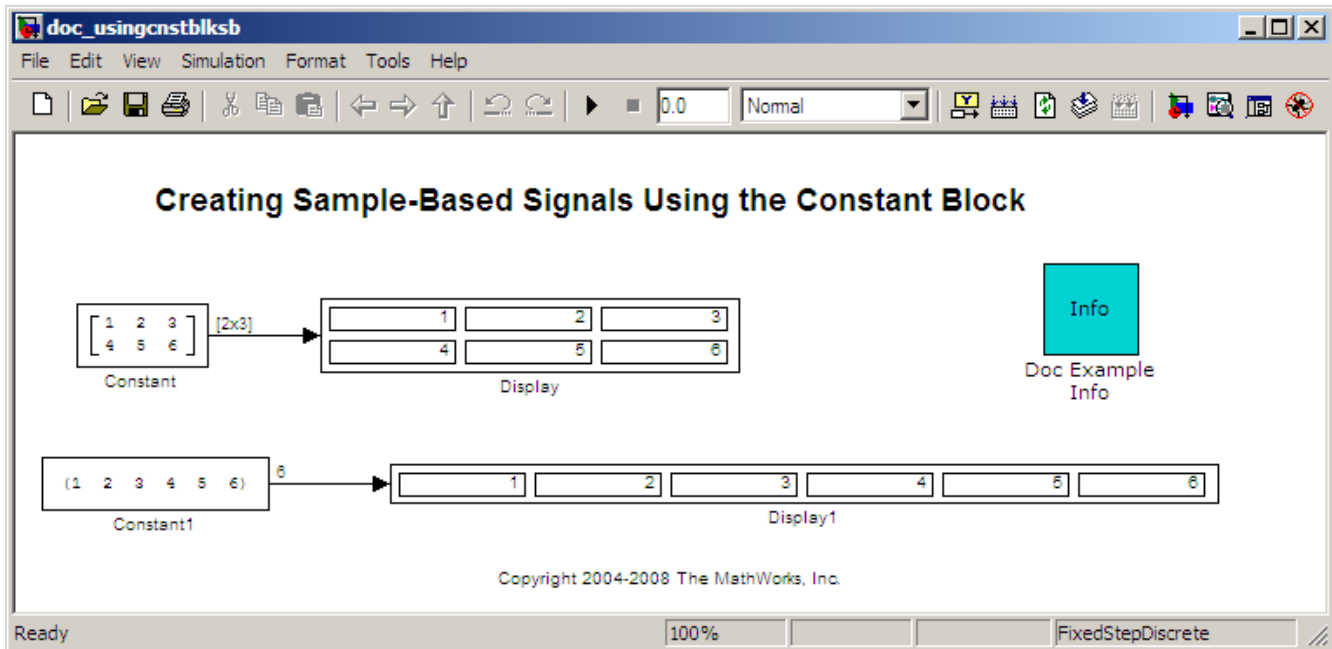
- 1 To add another sample-based signal to your model, copy the block diagram you created in the previous section and paste it below the existing sample-based signal in your model.
- 2 Double-click the Constant1 block, and set the block parameters as follows:

- **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 a Sample-Based Signal Using the Signal from Workspace Block

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

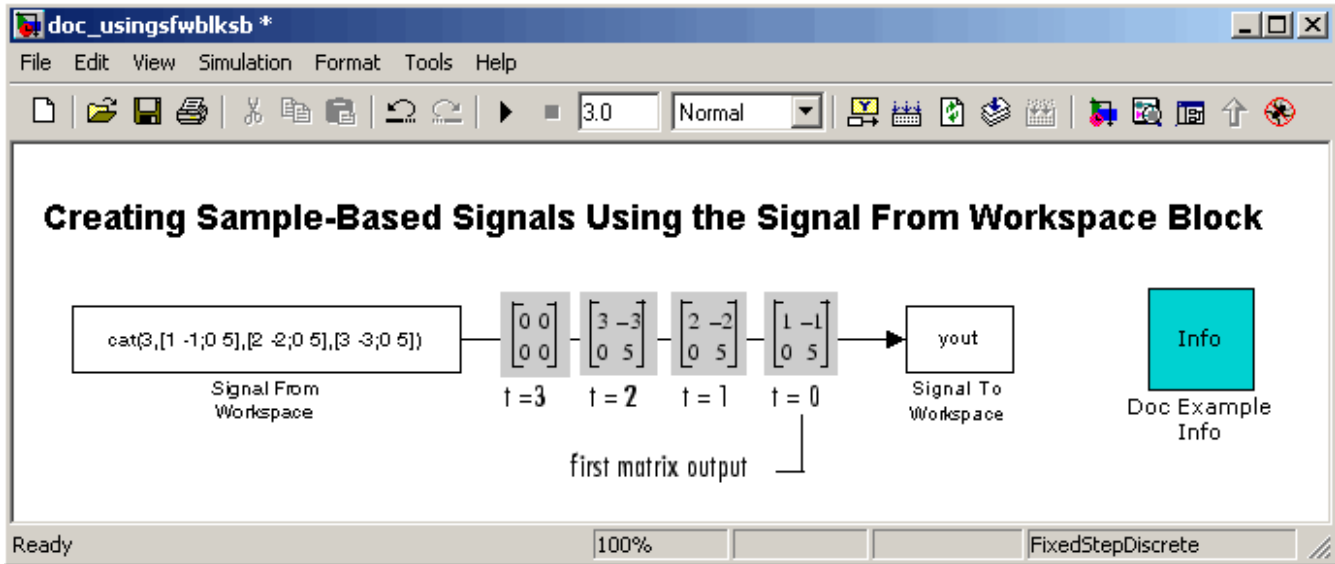
- 1** Create a new Simulink model.
- 2** From the Sources library, click-and-drag a Signal From Workspace block into the model.
- 3** From the Sinks library, click-and-drag a Signal To Workspace block into the model.
- 4** Connect the two blocks.
- 5** Double-click the Signal From Workspace block, and 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 sample-based 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 **Format** menu, point to **Port/Signal Displays**, 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.



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 sample-based signal with sample period of 1 second using the Signal From Workspace block.

Create Frame-Based Signals

Note Starting in R2010b, many DSP System Toolbox blocks received a new parameter to control whether they perform sample- or frame-based processing. The following content has not been updated to reflect this change. For more information, see the “Frame-Based Processing” section of the Release Notes.

In this section...

“Create a Frame-Based Signal Using the Sine Wave Block” on page 1-19

“Create a Frame-Based Signal Using the Signal from Workspace Block” on page 1-22

Create a Frame-Based Signal Using the Sine Wave Block

A frame-based signal is propagated through a model in batches of samples called frames. Frame-based processing can significantly improve the performance of your model by decreasing the amount of time it takes your simulation to run.

One of the most commonly used blocks in the Sources library is the Sine Wave block. This topic describes how to create a three-channel frame-based signal using the 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 Sinks library, click-and-drag a Signal 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

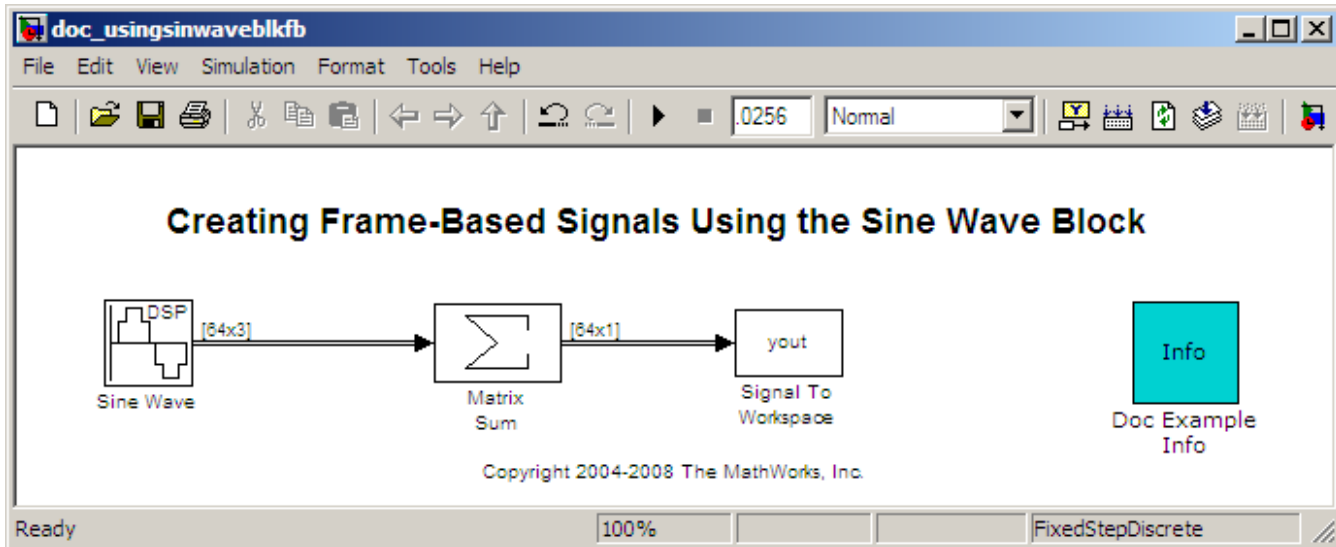
Based on these parameters, the Sine Wave block outputs three sinusoids with amplitudes 1, 3, and 2 and frequencies 100, 250, and 500 hertz, respectively. The sample period, 1/5000, is 10 times the highest sinusoid frequency, which satisfies the Nyquist criterion. The frame size is 64 for all sinusoids, 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 frame-based signal 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 **Format** menu, point to **Port/Signal Displays**, and select **Signal Dimensions**.
- 10** Run the model.

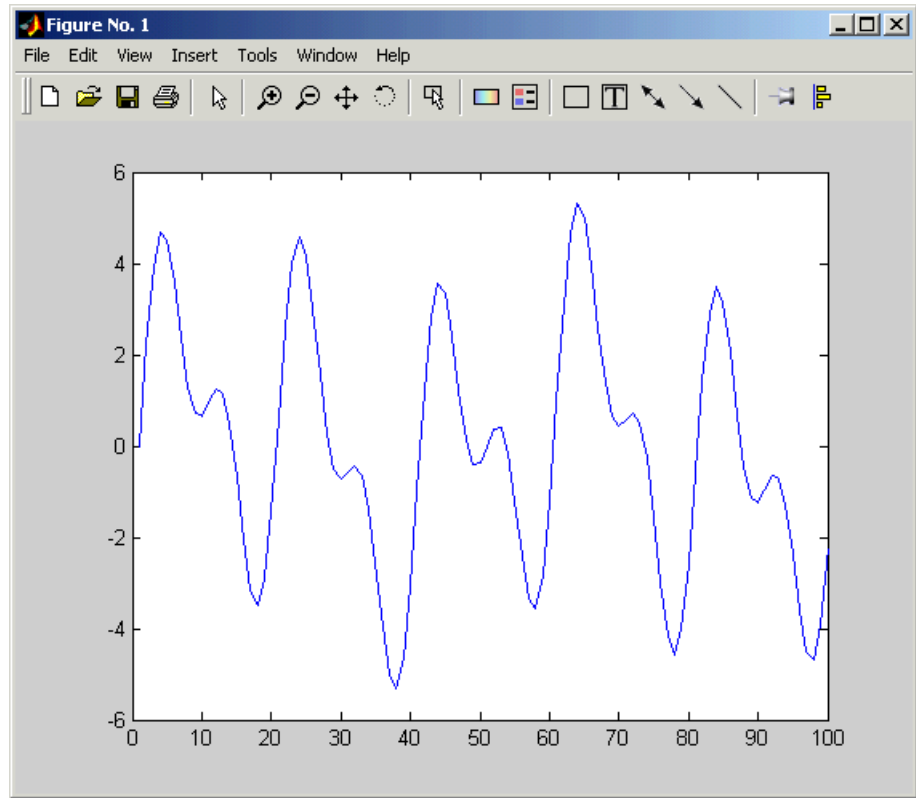
Your model should now look similar to the following figure. You can also open the model by typing `ex_usingsinwaveblkfb` at the MATLAB command line.



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 frame-based signal together and displayed the results in a figure window.

Create a Frame-Based Signal Using the Signal from Workspace Block

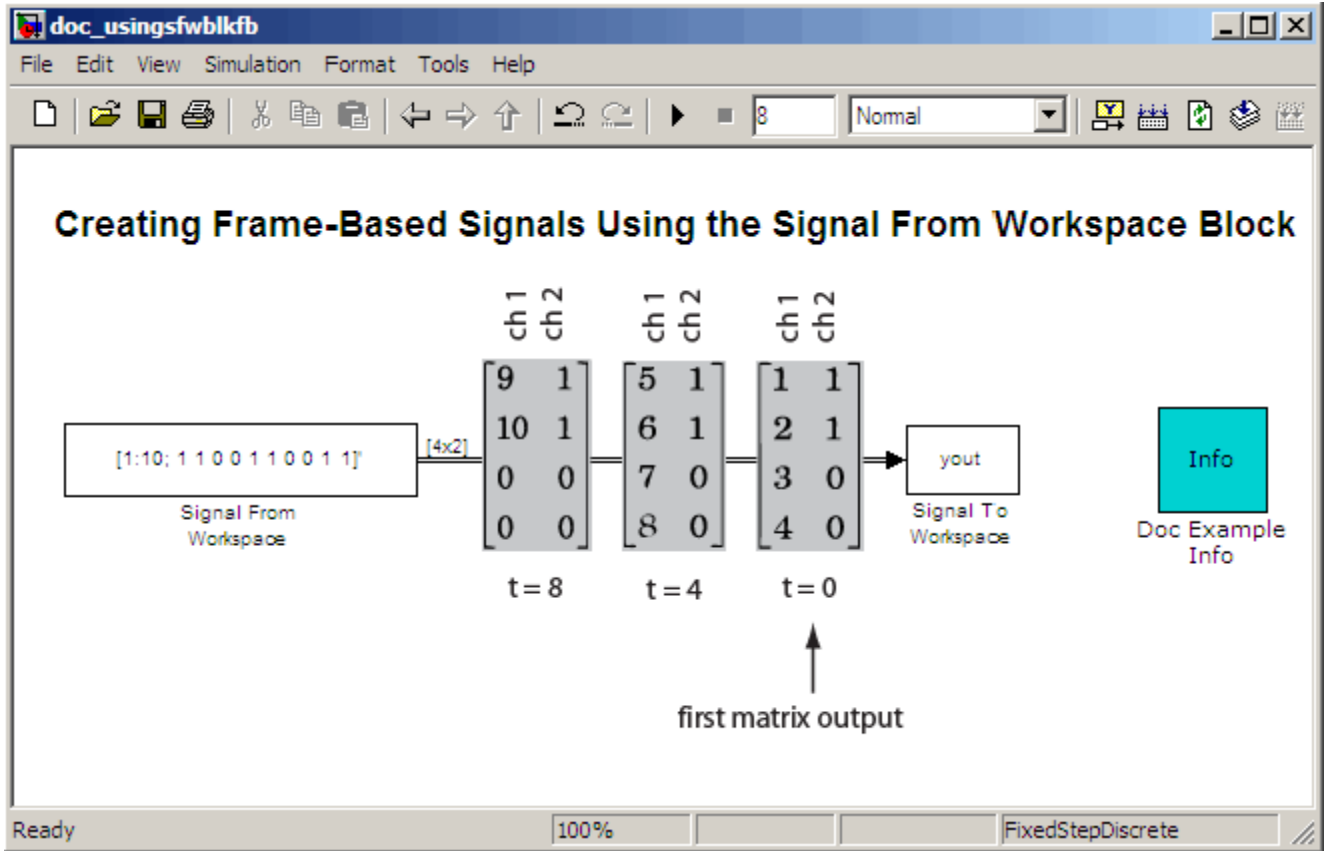
A frame-based signal is propagated through a model in batches of samples called frames. Frame-based processing can significantly improve the performance of your model by decreasing the amount of time it takes your simulation to run. This topic describes how to create a two-channel frame-based signal 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 Sinks library, click-and-drag a Signal 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, frame-based signal has 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 **Format** menu, point to **Port/Signal Displays**, 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 frame-based signal and exported it to the MATLAB workspace.

Create Multichannel Sample-Based Signals

In this section...
“Multichannel Sample-Based Signals” on page 1-26
“Create a Multichannel Sample-Based Signal by Combining Single-Channel Sample-Based Signals” on page 1-26
“Create a Multichannel Sample-Based Signal by Combining Multichannel Sample-Based Signals” on page 1-29

Multichannel Sample-Based Signals

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 sample-based signal with $M \times N$ channels is represented by a sequence of M -by- N matrices. Multiple sample-based signals can be combined into a single multichannel sample-based signal using the Concatenate block. In addition, several multichannel sample-based signals can be combined into a single multichannel sample-based signal using the same technique.

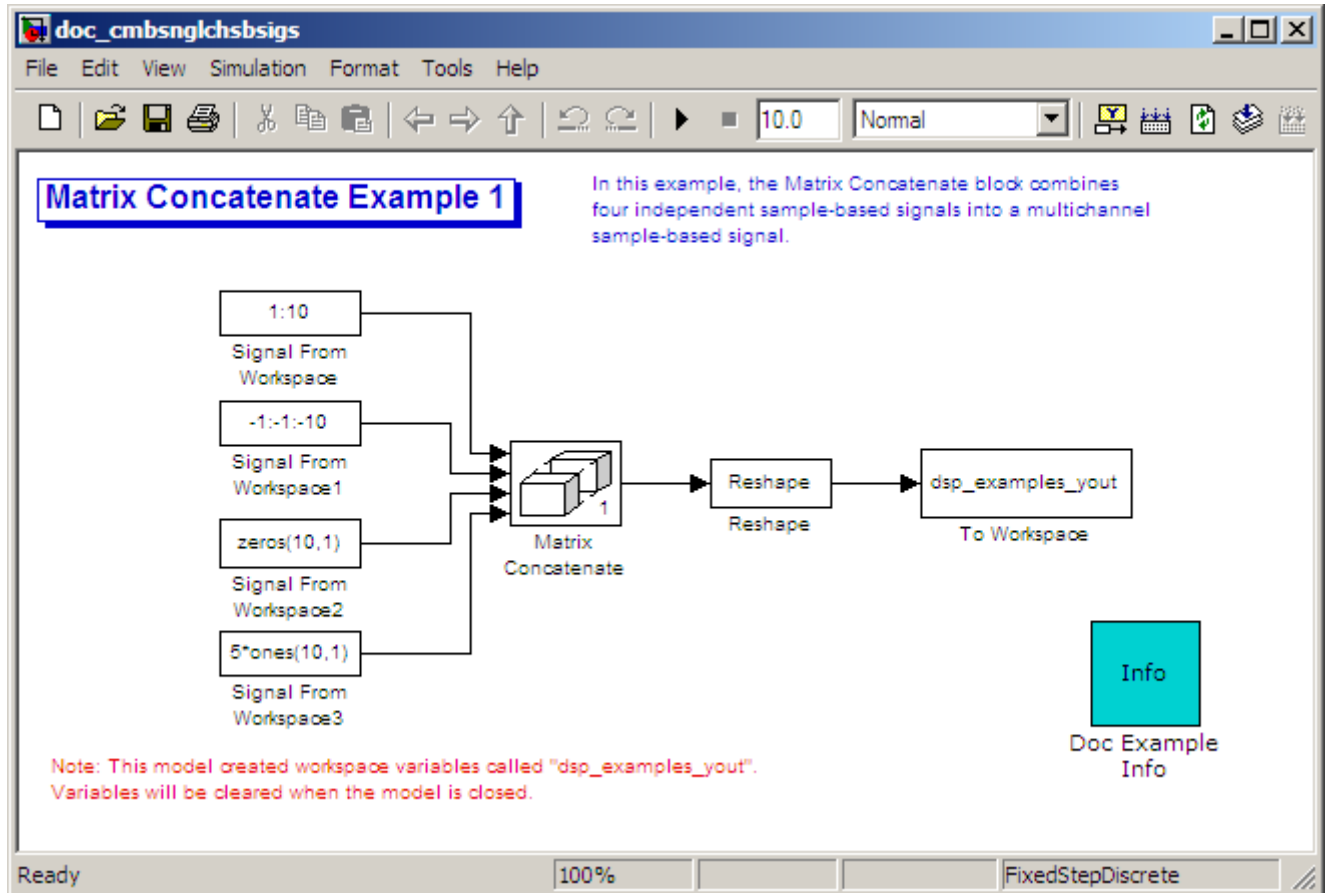
Create a Multichannel Sample-Based Signal by Combining Single-Channel Sample-Based Signals

You can combine individual sample-based 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_cmbsinglechsbsigs
```

at the MATLAB command line.



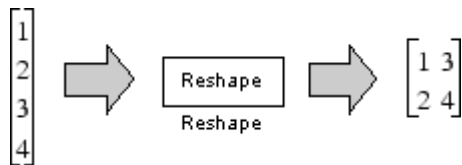
- 2 Double-click the Signal From Workspace block, and set the **Signal** parameter to 1:10. Click **OK**.
- 3 Double-click the Signal From Workspace1 block, and set the **Signal** parameter to -1:-1:-10. Click **OK**.
- 4 Double-click the Signal From Workspace2 block, and set the **Signal** parameter 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 sample-based 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 columnwise, 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 sample-based 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, sample-based 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.

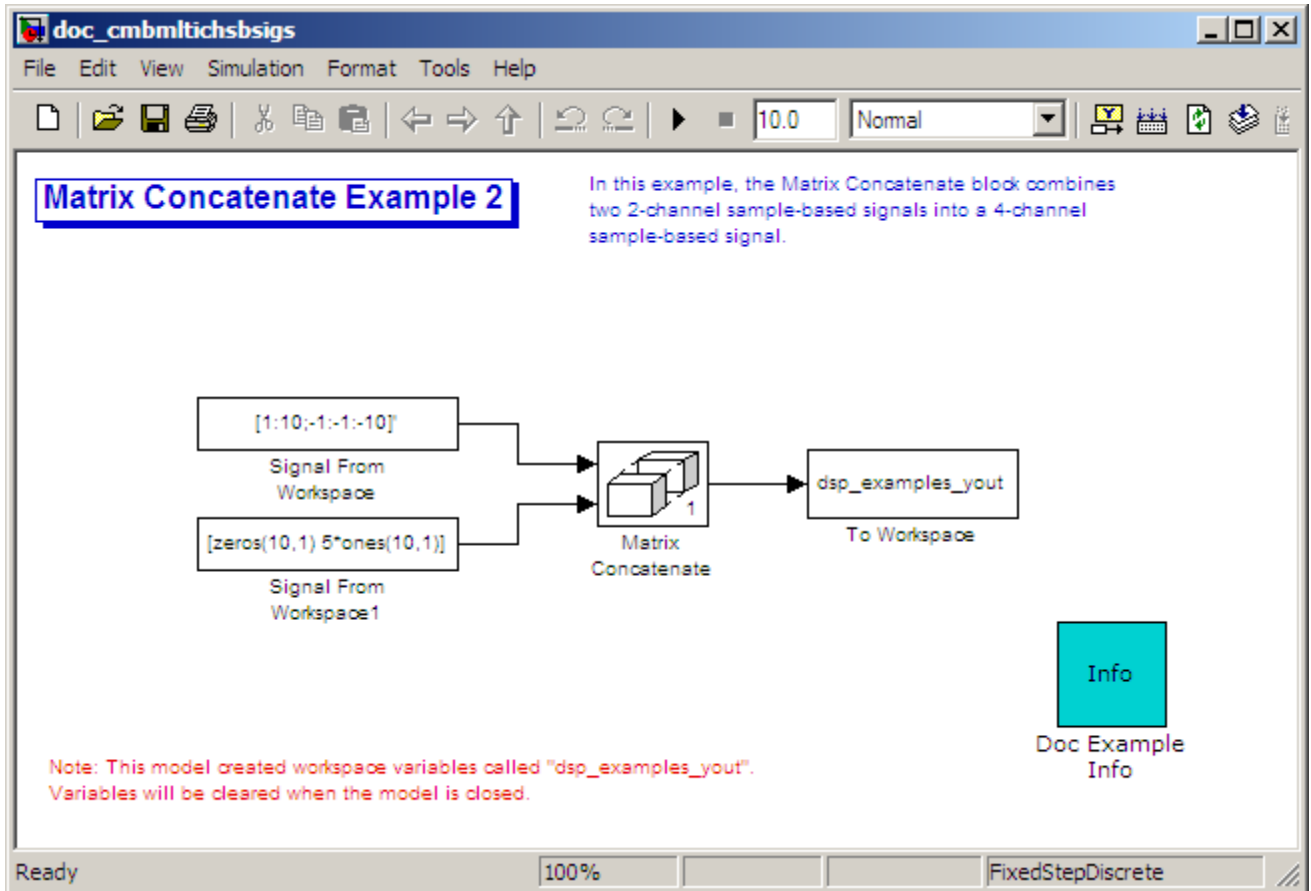
Create a Multichannel Sample-Based Signal by Combining Multichannel Sample-Based Signals

You can combine existing multichannel sample-based 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 $[\text{zeros}(10,1) \ 5*\text{ones}(10,1)]$. 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 sample-based 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.

Create Multichannel Frame-Based Signals

In this section...

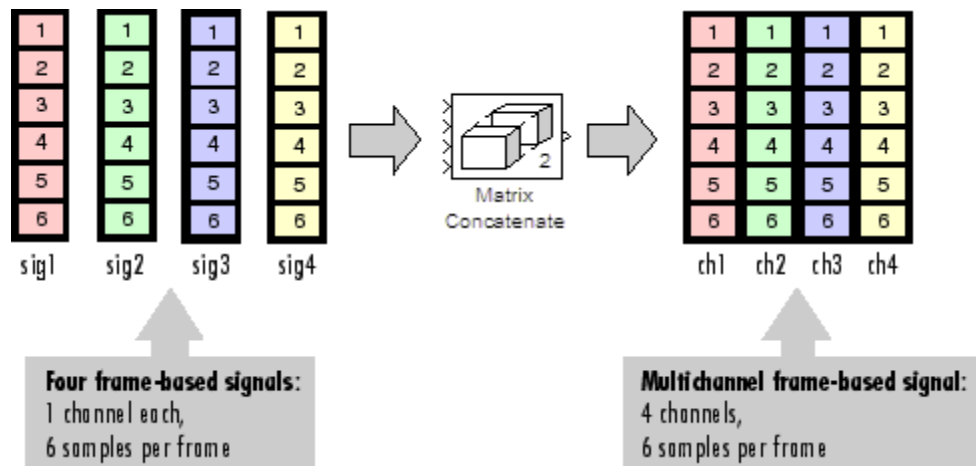
“Multichannel Frame-Based Signals” on page 1-32

“Create a Multichannel Frame-Based Signal Using the Concatenate Block” on page 1-33

Multichannel Frame-Based Signals

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 frame-based signal with N channels and frame size M is represented by a sequence of M -by- N matrices. Multiple individual frame-based signals, with the same frame rate and size, can be combined into a multichannel frame-based signal using the Simulink Matrix Concatenate block. Individual signals can be added to an existing multichannel signal in the same way.



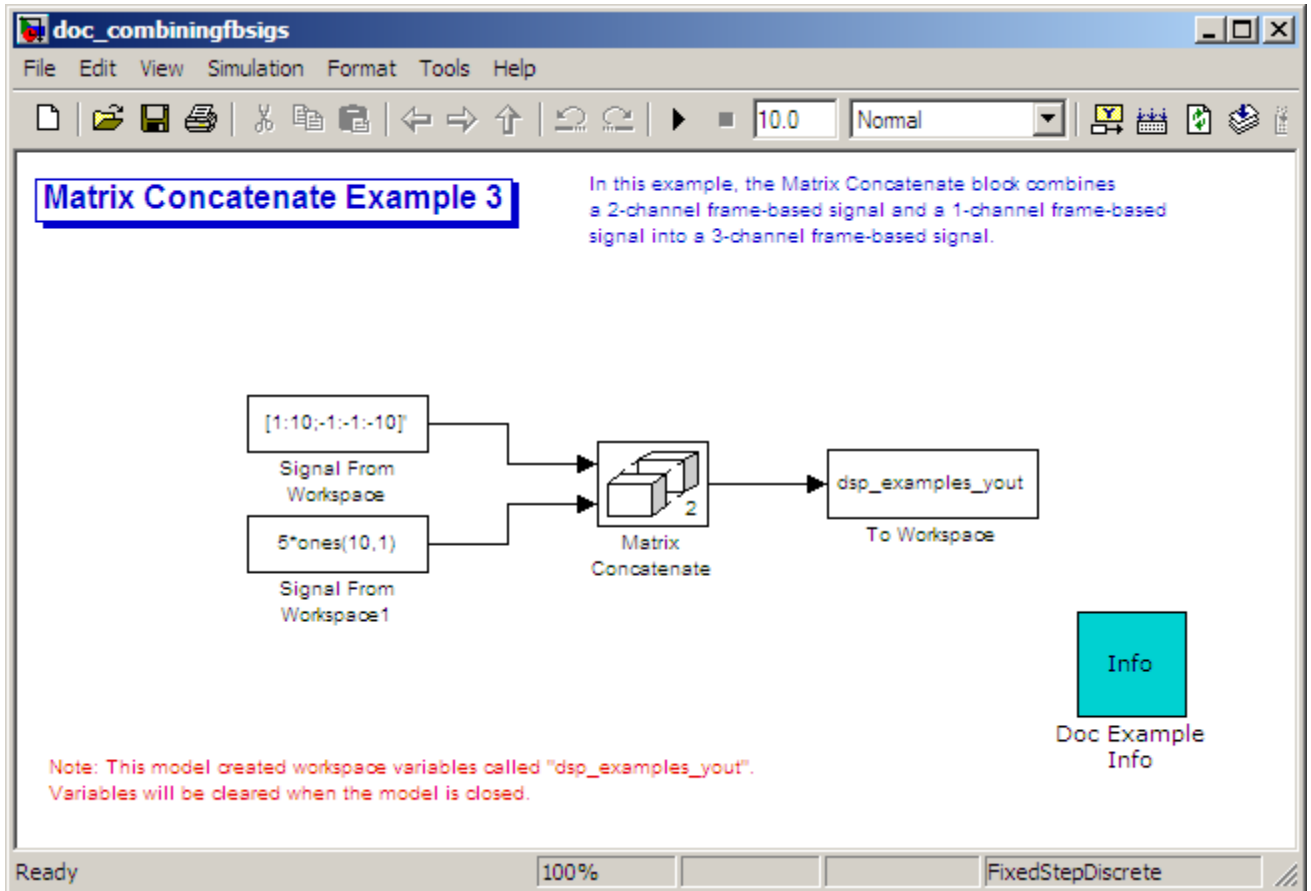
Create a Multichannel Frame-Based Signal Using the Concatenate Block

You can combine existing frame-based 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 frame-based signal is combined with a two-channel frame-based signal to produce a three-channel frame-based 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 frame-based 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. When you combine frame-based signals into multichannel signals, the original signals must have the same frame rate and 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.

Deconstruct Multichannel Sample-Based Signals

Note Starting in R2010b, many DSP System Toolbox blocks received a new parameter to control whether they perform sample- or frame-based processing. The following content has not been updated to reflect this change. For more information, see the “Frame-Based Processing” section of the Release Notes.

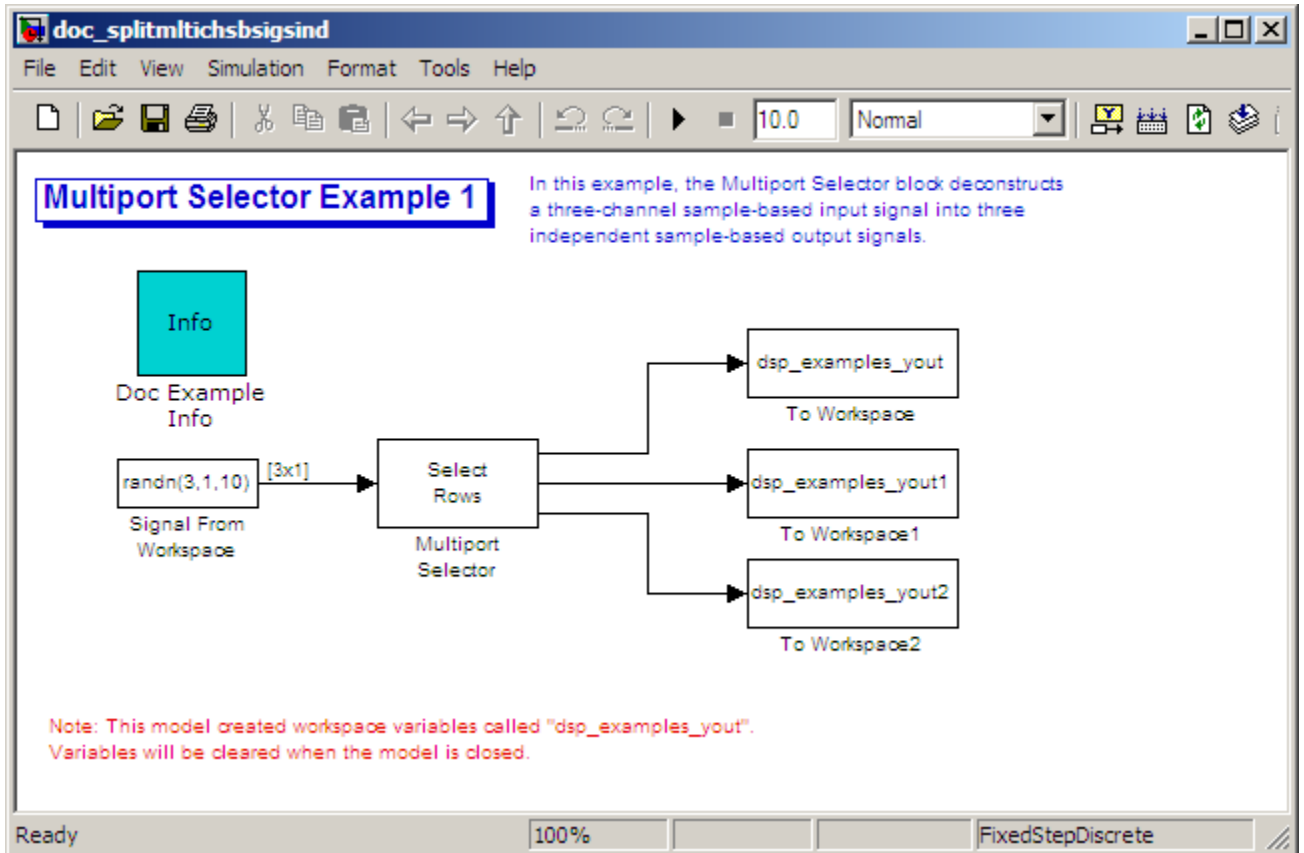
In this section...
“Split Multichannel Sample-Based Signals into Individual Signals” on page 1-36
“Split Multichannel Sample-Based Signals into Several Multichannel Signals” on page 1-39

Split Multichannel Sample-Based 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 sample-based signal into single-channel sample-based 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 sample-based signal is deconstructed into three independent sample-based signals:

- 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, sample-based 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 sample-based 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.

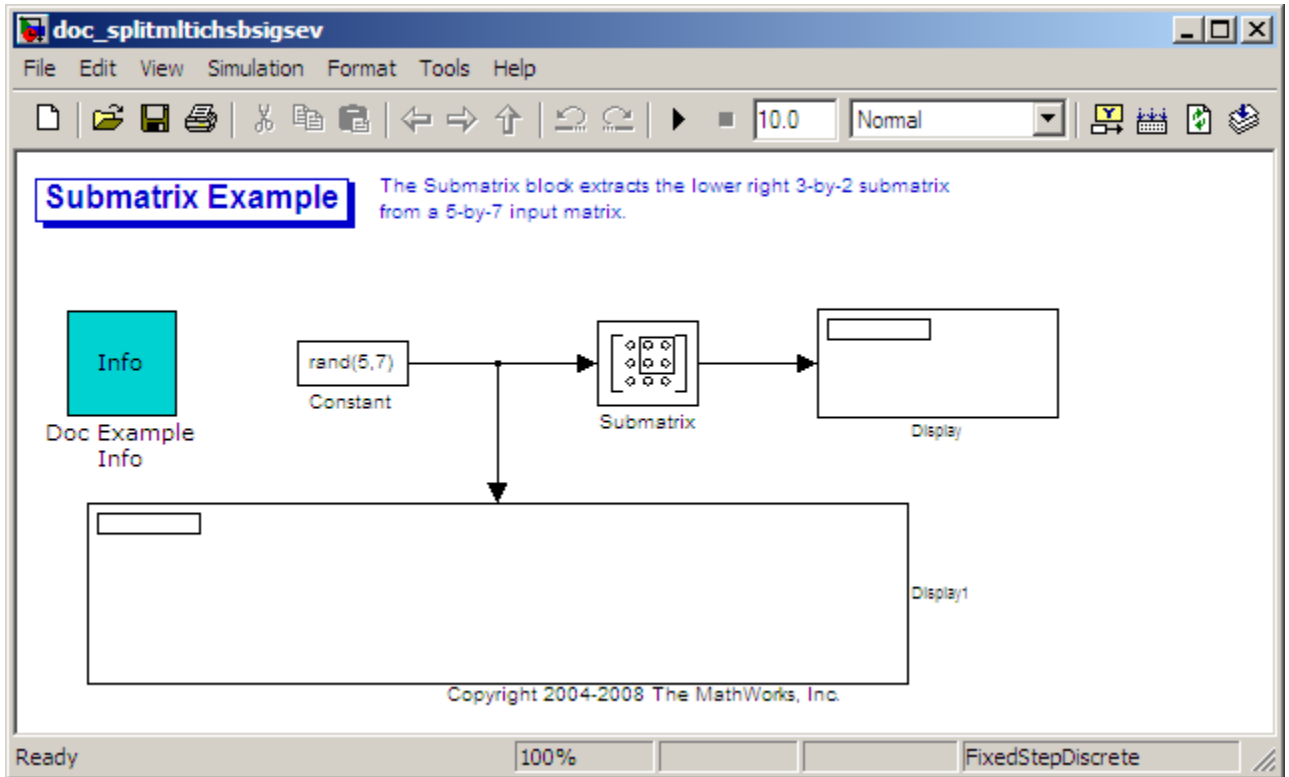
You have now successfully created three, single-channel sample-based signals from a multichannel sample-based signal using a Multiport Selector block.

Split Multichannel Sample-Based 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, Multipoint Selector, and Submatrix blocks.

You can split a multichannel sample-based signal into other multichannel sample-based 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 sample-based signal. In this example, you extract a six-channel, sample-based signal from a 35-channel, sample-based signal (5-by-7 matrix):

- 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** = $\text{rand}(5,7)$
- **Interpret vector parameters as 1-D** = Clear this check box
- **Sampling mode** = Sample based
- **Sample Time** = 1

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

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

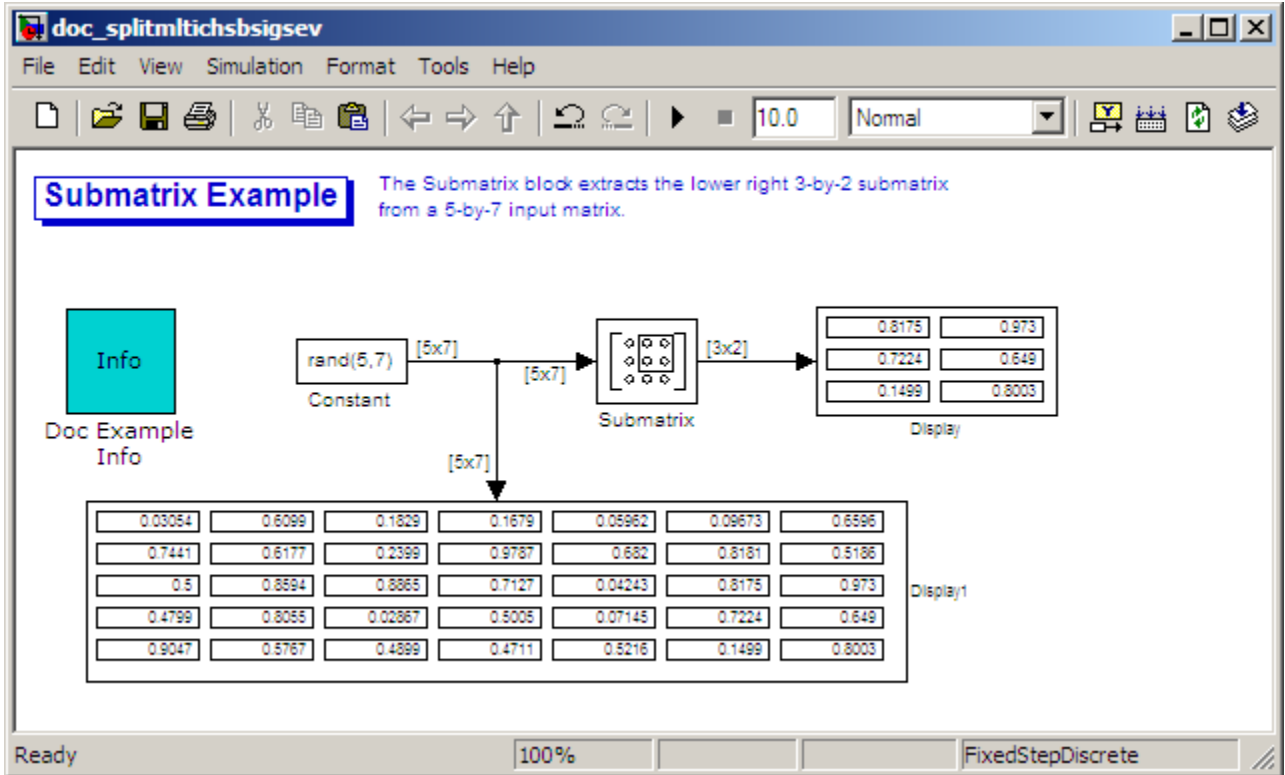
4 Double-click the Submatrix block. Set the block parameters as follows, and then click **OK**:

- **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, sample-based signal from a 35-channel sample-based signal using a Submatrix block.

Deconstruct Multichannel Frame-Based Signals

Note Starting in R2010b, many DSP System Toolbox blocks received a new parameter to control whether they perform sample- or frame-based processing. The following content has not been updated to reflect this change. For more information, see the “Frame-Based Processing” section of the Release Notes.

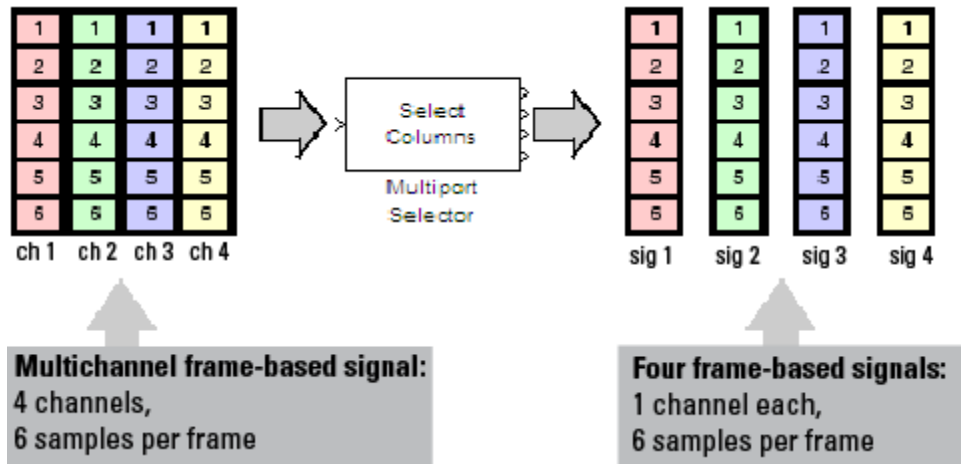
In this section...
“Split Multichannel Frame-Based Signals into Individual Signals” on page 1-43
“Reorder Channels in Multichannel Frame-Based Signals” on page 1-48

Split Multichannel Frame-Based 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 frame-based signal. These signals form single-channel frame-based signals that have the same frame rate and size of the multichannel signal.

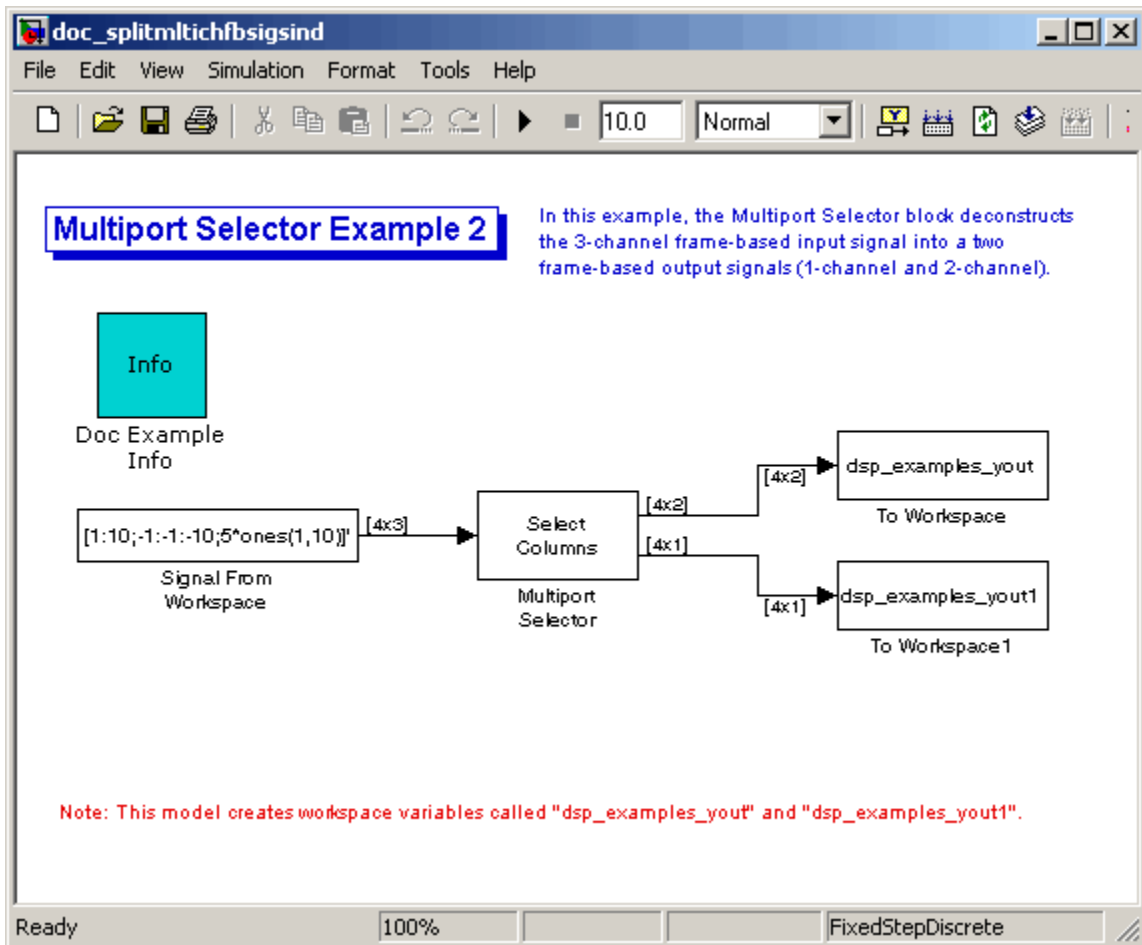
The figure below is a graphical representation of this process.



In this example, you use the Multiport Selector block to extract a single-channel and a two channel frame-based signal from a multichannel frame-based signal:

- 1 Open the Multiport Selector Example 2 model by typing `ex_splitmlichfbsigsind`

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, frame-based signal with a frame size of four.

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

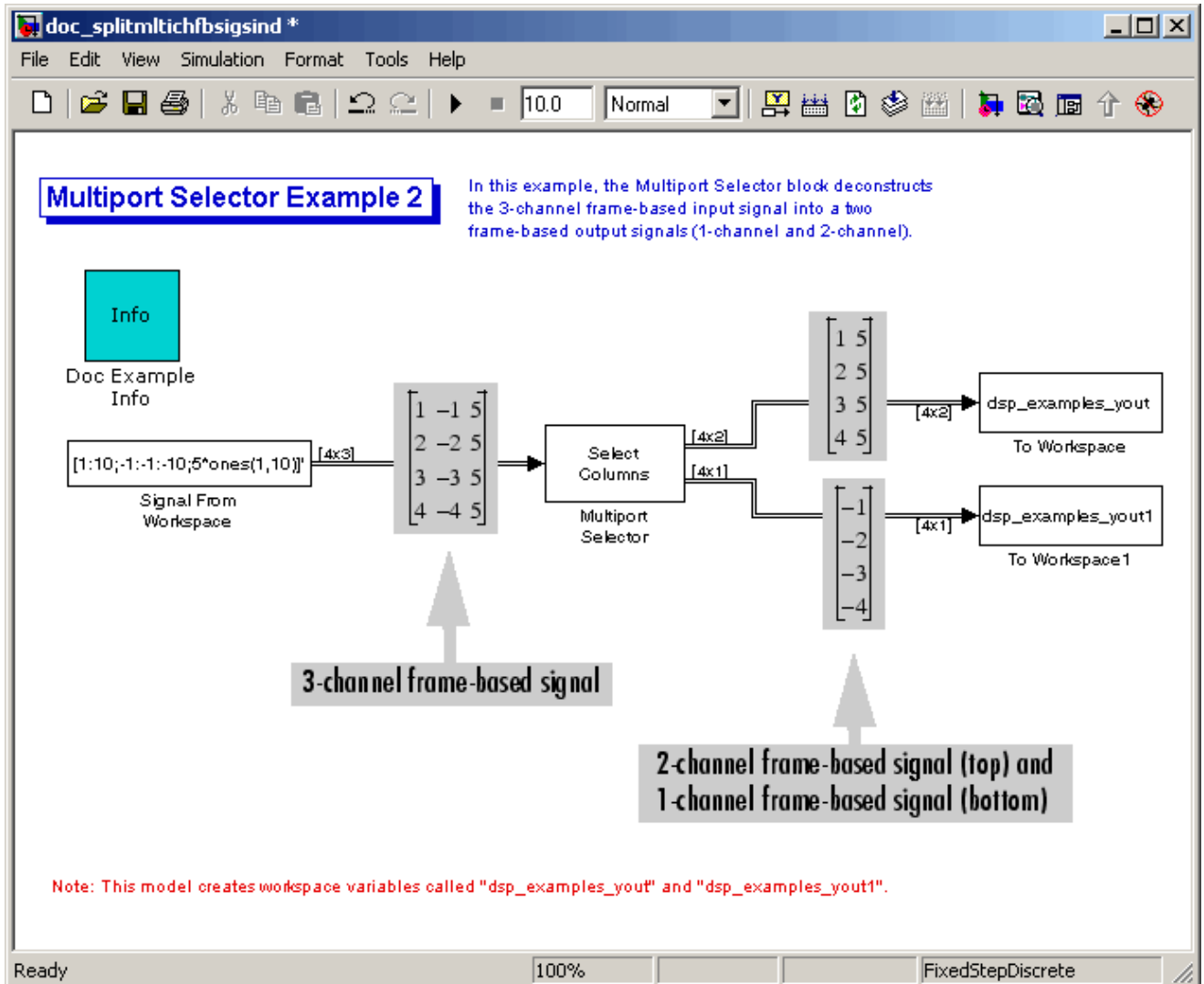
4 Double-click the Multiport Selector block. Set the block parameters as follows, and then click **OK**:

- **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 frame-based signal into a single-channel signal and a two-channel signal.



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

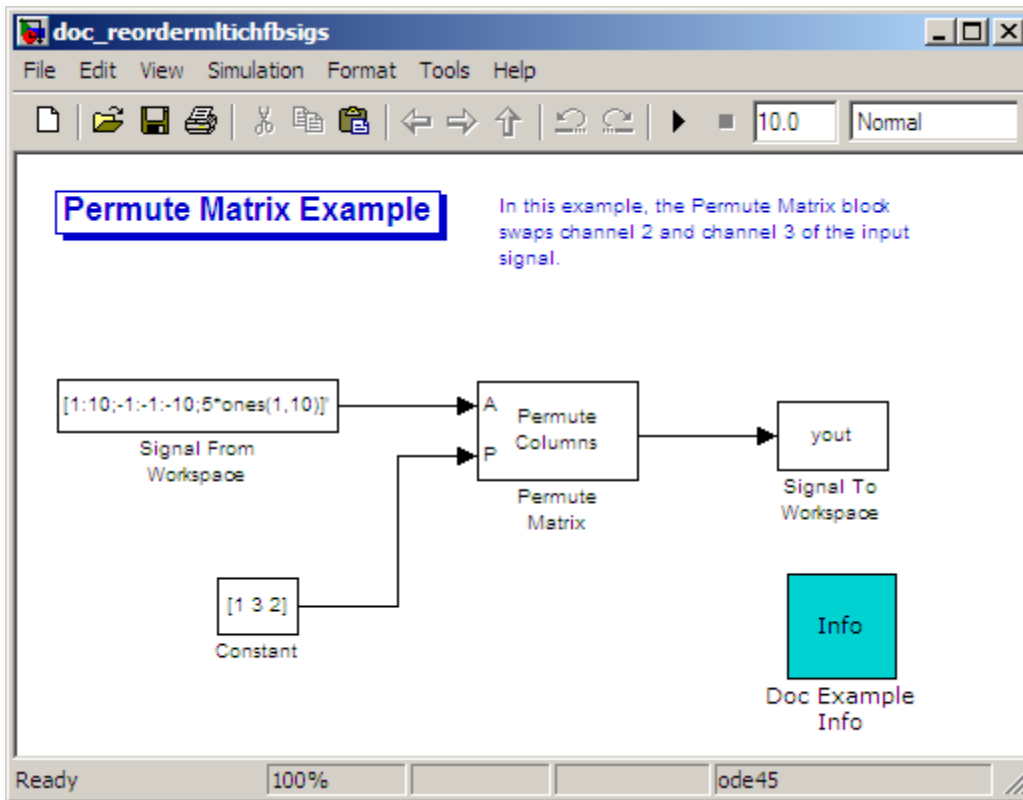
You have now successfully created a single-channel and a two-channel frame-based signal from a multichannel frame-based signal using the Multiport Selector block.

Reorder Channels in Multichannel Frame-Based 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-based 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 frame-based 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** = $[1:10;-1:-1:-10;5*\text{ones}(1,10)]'$
- **Sample time** = 1
- **Samples per frame** = 4

Based on these parameters, the Signal From Workspace block outputs a three-channel, frame-based 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
- **Sampling mode** = Frame based
- **Frame period** = 4

The discrete-time, frame-based 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.

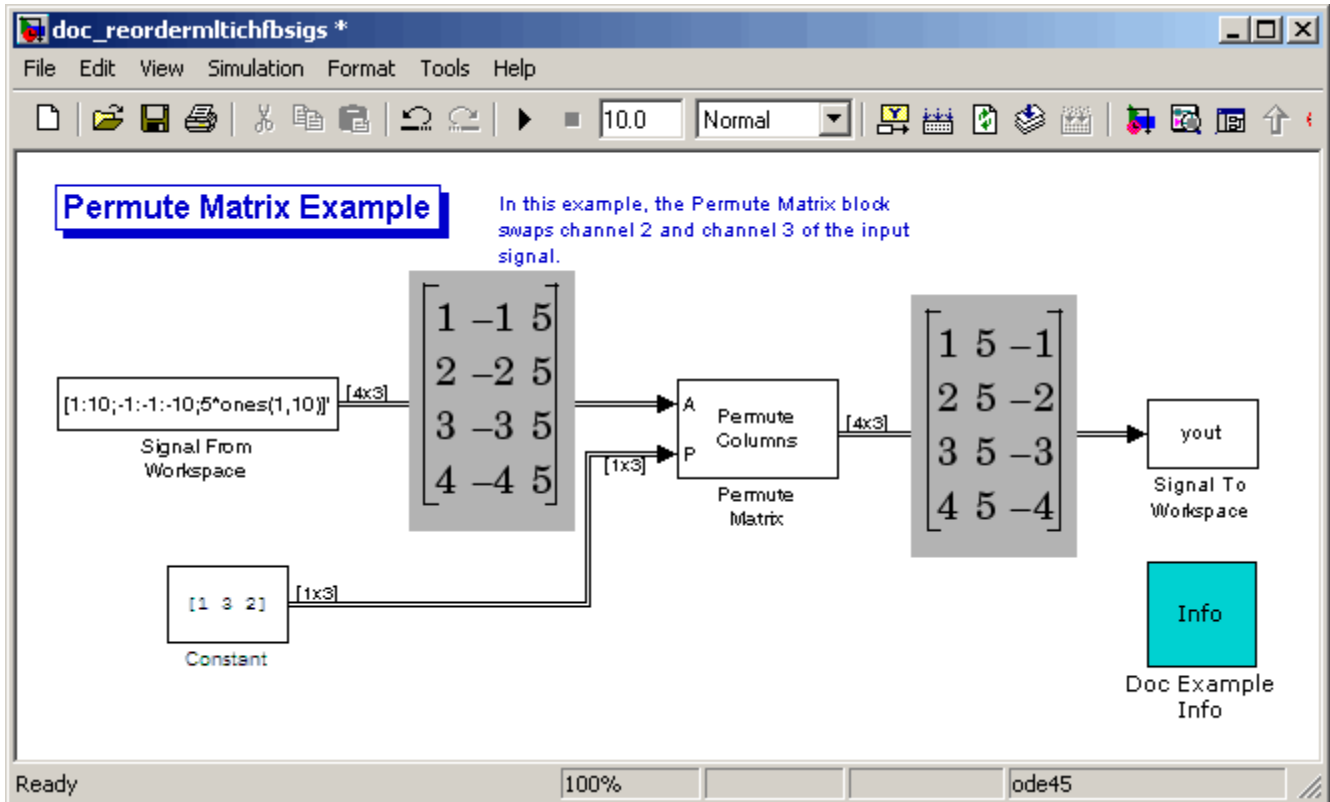
5 Double-click the Permute Matrix block. Set the block parameters as follows, and then click **OK**:

- **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.



The second and third channel of the frame-based 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-based signal using the Permute Matrix block.

Import and Export Sample-Based Signals

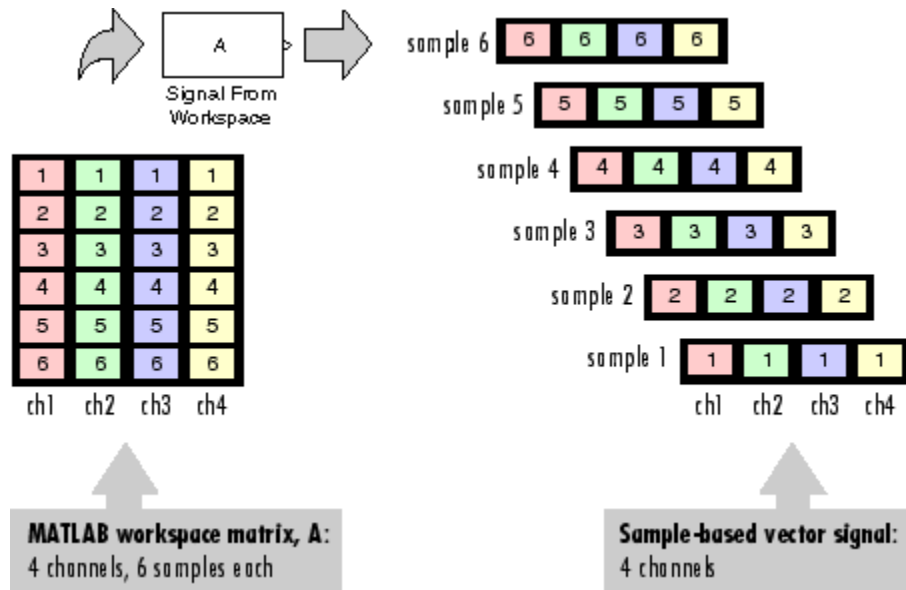
Note Starting in R2010b, many DSP System Toolbox blocks received a new parameter to control whether they perform sample- or frame-based processing. The following content has not been updated to reflect this change. For more information, see the “Frame-Based Processing” section of the Release Notes.

In this section...
“Import Sample-Based Vector Signals” on page 1-52
“Import Sample-Based Matrix Signals” on page 1-55
“Export Sample-Based Signals” on page 1-59

Import Sample-Based Vector Signals

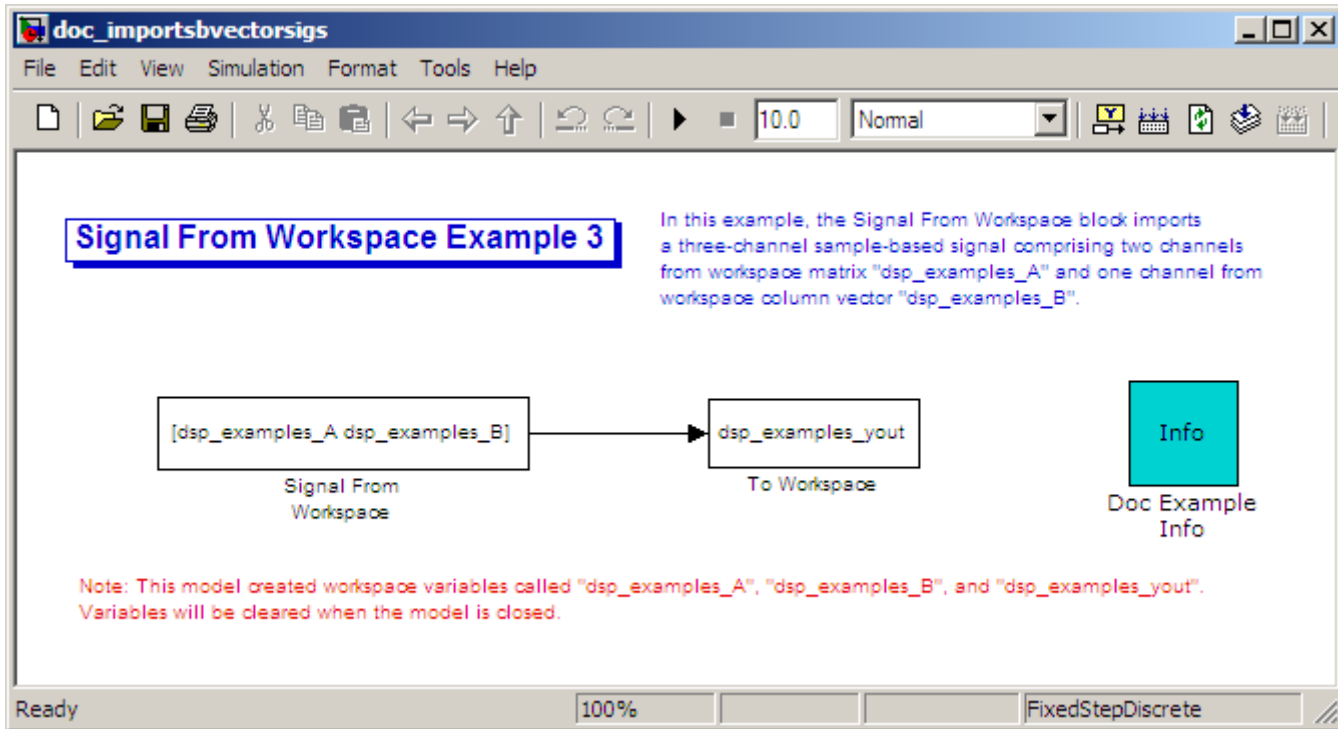
The Signal From Workspace block generates a sample-based vector signal when the variable or expression in the **Signal** parameter is a matrix and the **Samples per frame** parameter is set to 1. Each column of the input matrix represents a different channel. Beginning with the first row of the matrix, 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 a sample-based 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:

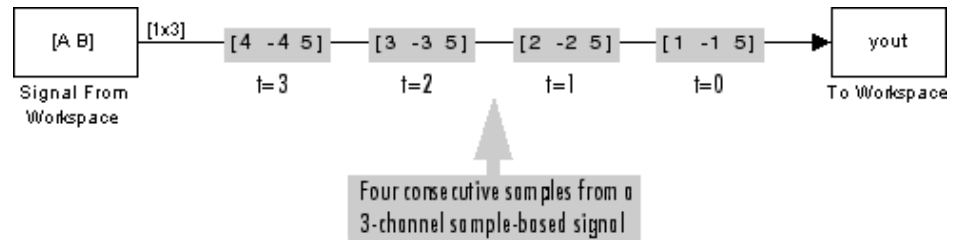
- **Signal** = [A B]
- **Sample time** = 1
- **Samples per frame** = 1
- **Form output after final data value** = Setting to zero

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 sample-based 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.



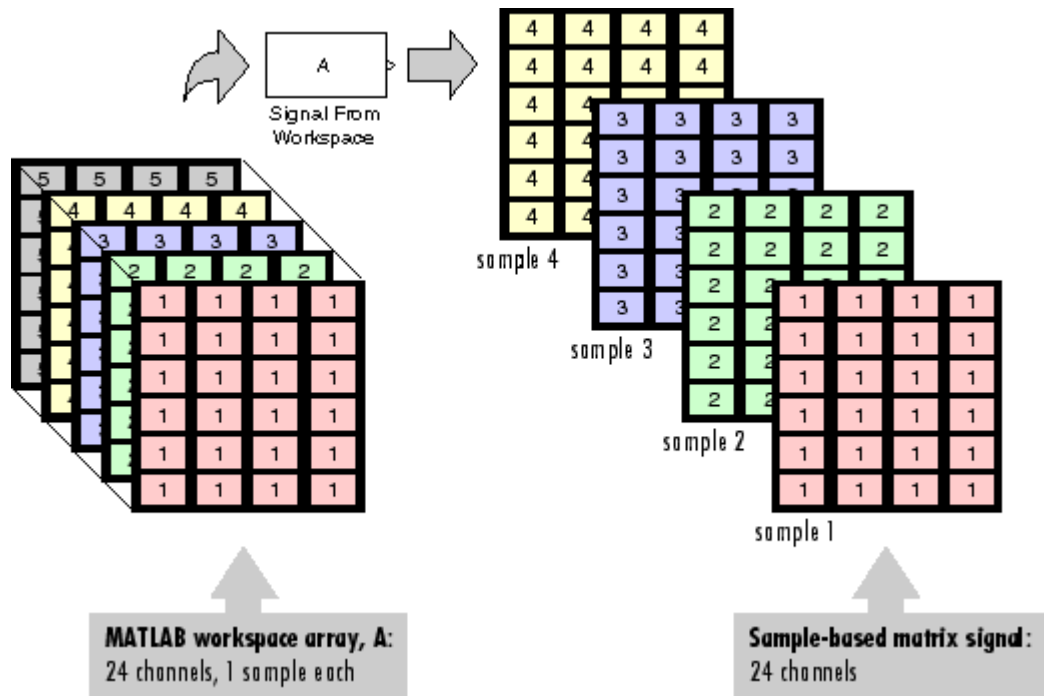
The first row of the input matrix `[A B]` is output at time $t=0$, the second row of the input matrix is output at time $t=1$, and so on.

You have now successfully imported a sample-based vector signal into your signal processing model using the Signal From Workspace block.

Import Sample-Based Matrix Signals

The Signal From Workspace block generates a sample-based matrix signal when the variable or expression in the **Signal** parameter is a three-dimensional array and the **Samples per frame** parameter is set to 1. 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 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, sample-based matrix signal into a Simulink model:

- 1 Open the Signal From Workspace Example 4 model by typing `ex_importsbmatrixsigs` at the MATLAB command line.

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".

```

    graph LR
      A["dsp_examples_A  
Signal From  
Workspace"] -- "[2x2]" --> B["dsp_examples_yout  
To Workspace"]
  
```

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"
  
```

Info
Doc Example Info

Ready 100% FixedStepDiscrete

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, sample-based 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

Info

Doc Example Info

Ready 100% FixedStepDiscrete

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

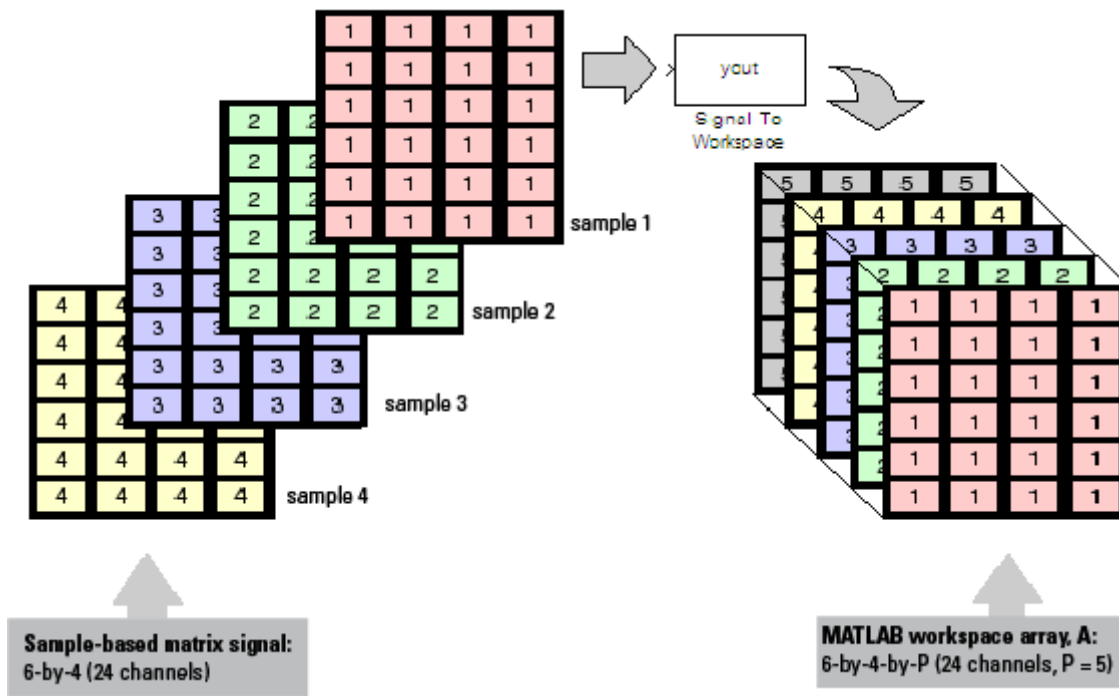
You have now successfully imported a sample-based matrix signal into your model using the Signal From Workspace block.

Export Sample-Based Signals

The Signal 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 sample-based signal, with $M \times N$ channels, is represented in Simulink as a sequence of M -by- N matrices. When the input to the Signal To Workspace block is a sample-based signal, 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.

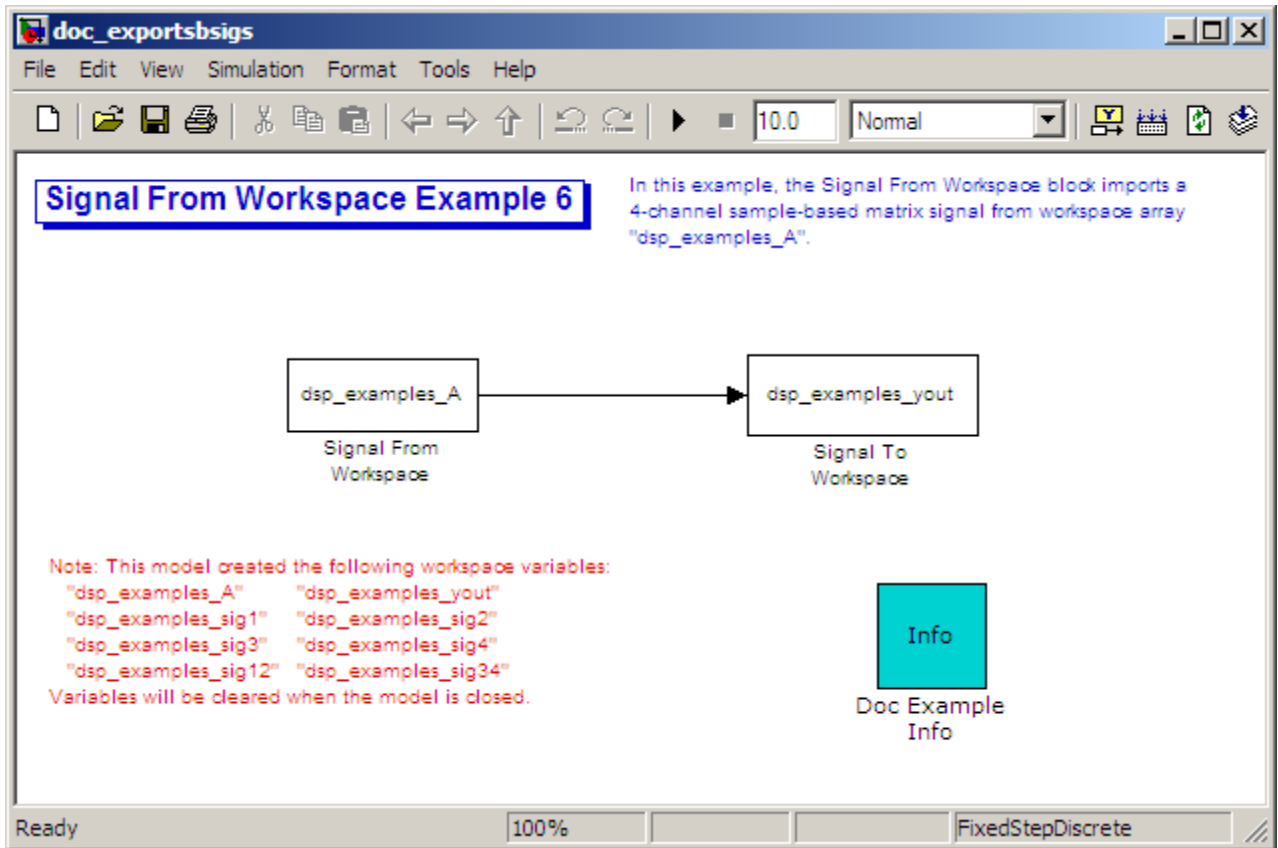
The following figure is the graphical illustration of this process using a 6-by-4 sample-based signal exported to workspace array A .



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 Signal To Workspace block to export a sample-based 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:

<code>dsp_examples_A</code>	2x2x100	3200	double array
<code>dsp_examples_sig1</code>	1x1x100	800	double array
<code>dsp_examples_sig12</code>	1x2x100	1600	double array
<code>dsp_examples_sig2</code>	1x1x100	800	double array
<code>dsp_examples_sig3</code>	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 sample-based signal called `dsp_examples_A`. This signal is then exported to the MATLAB workspace using a Signal 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 sample-based 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 Signal 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 Signal To Workspace block exports its sample-based 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 sample-based 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 sample-based signal from a Simulink model to the MATLAB workspace using the Signal To Workspace block.

Import and Export Frame-Based Signals

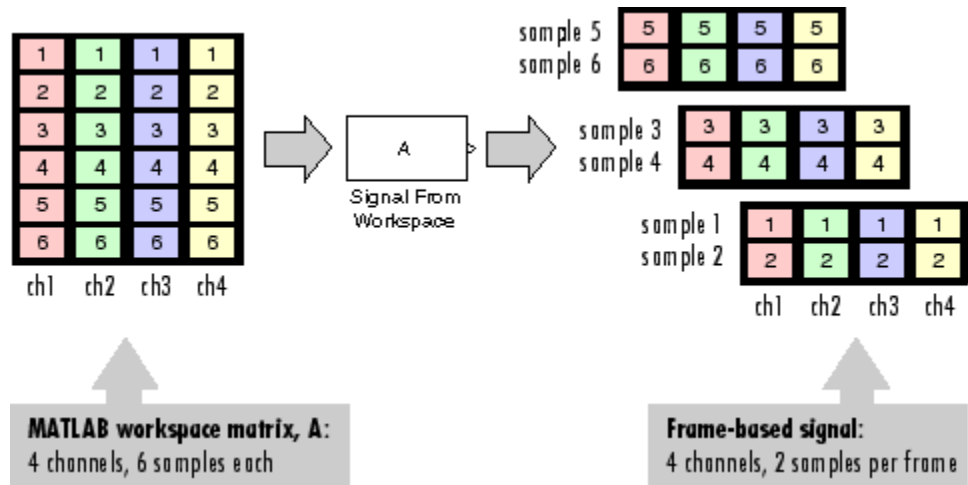
Note Starting in R2010b, many DSP System Toolbox blocks received a new parameter to control whether they perform sample- or frame-based processing. The following content has not been updated to reflect this change. For more information, see the “Frame-Based Processing” section of the Release Notes.

In this section...
“Import Frame-Based Signals” on page 1-64
“Export Frame-Based Signals” on page 1-67

Import Frame-Based Signals

The Signal From Workspace block creates a frame-based multichannel signal 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 $M \cdot T_s$ 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.



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-based signal and import it into the model:

- 1 Open the Signal From Workspace Example 5 model by typing

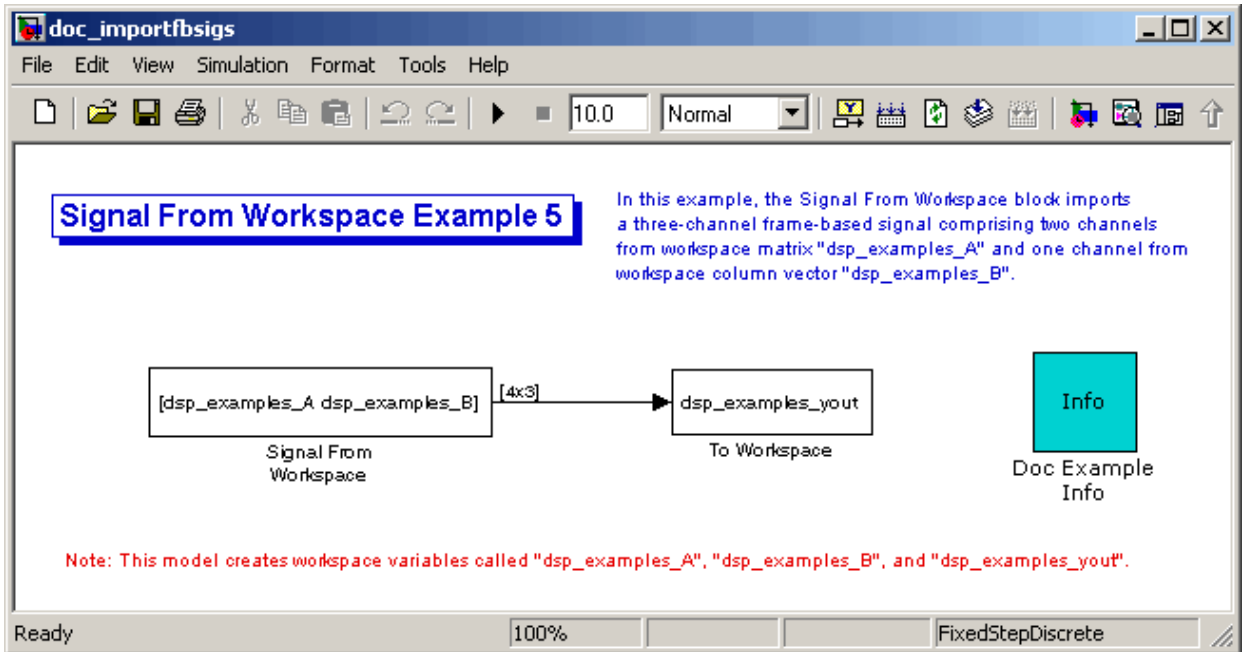
```
ex_importfbsigs
```

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:



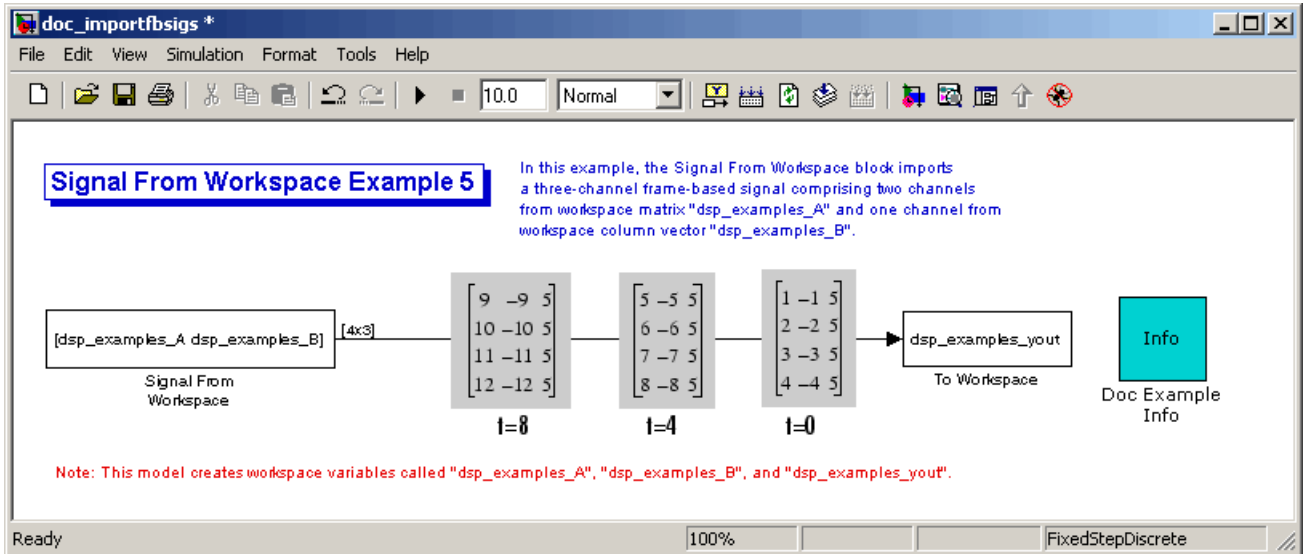
2 Double-click the Signal From Workspace block. Set the block parameters as follows, and then click **OK**:

- **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 frame-based 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-based signal is imported into your model.



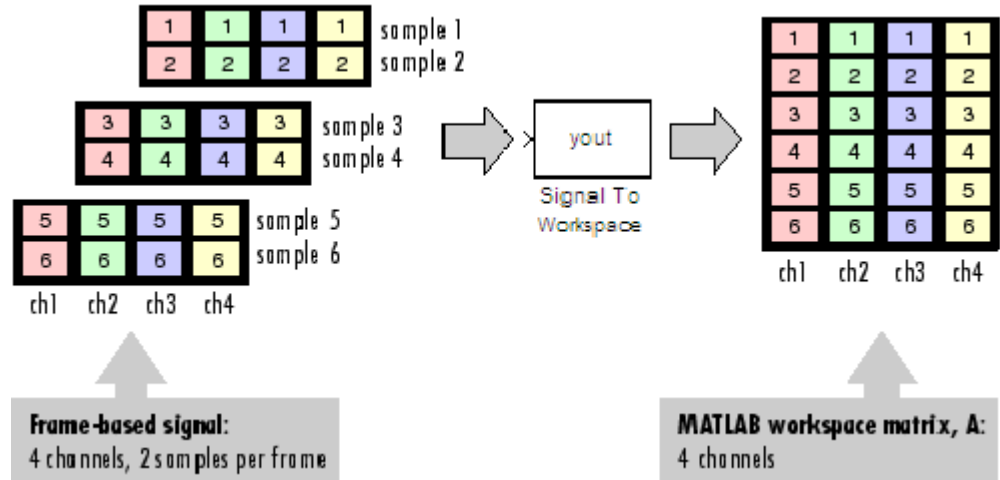
You have now successfully imported a three-channel frame-based signal into your model using the Signal From Workspace block.

Export Frame-Based Signals

The Signal 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.

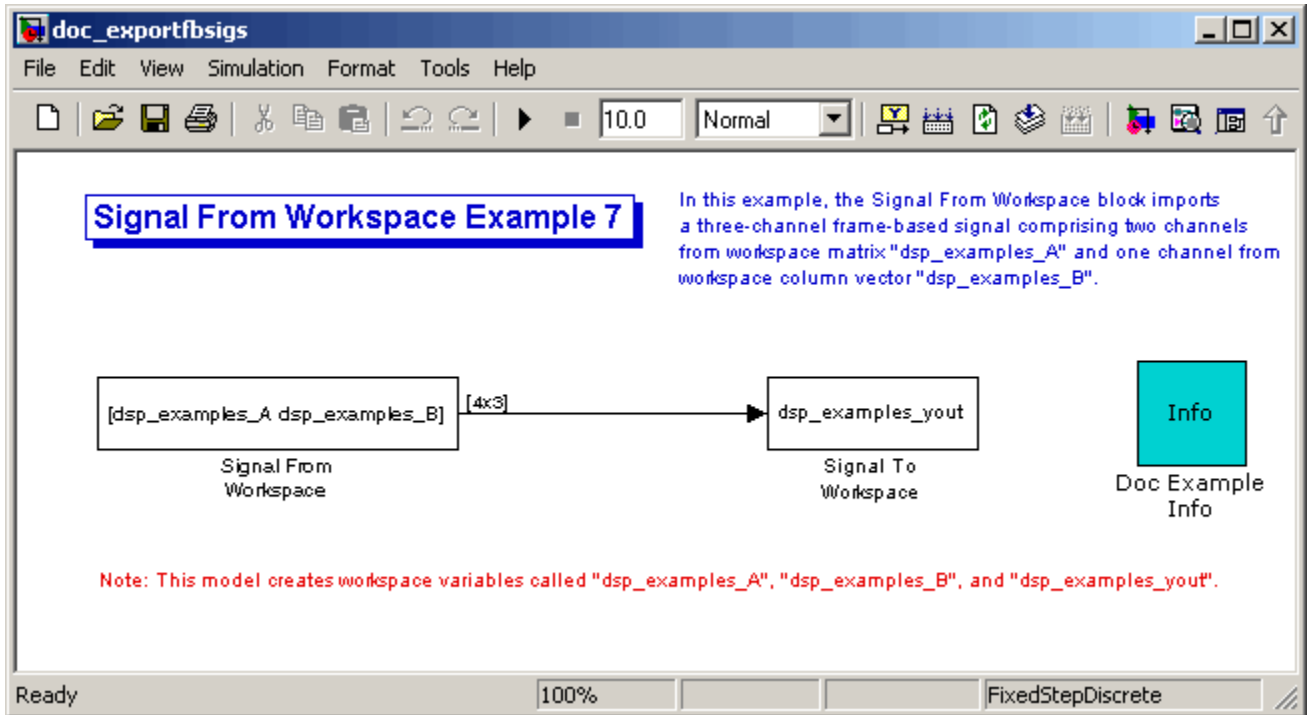
A frame-based signal with N channels and frame size M is represented by a sequence of M -by- N matrices. When the input to the Signal To Workspace block is a frame-based signal, the block creates a P -by- N array in the MATLAB workspace containing the P most recent samples from each channel. The number 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 frame-based signal with a frame size of 2 that is exported to matrix A in the MATLAB workspace.



In the following example, you use a Signal To Workspace block to export a frame-based signal 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 frame-based 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.

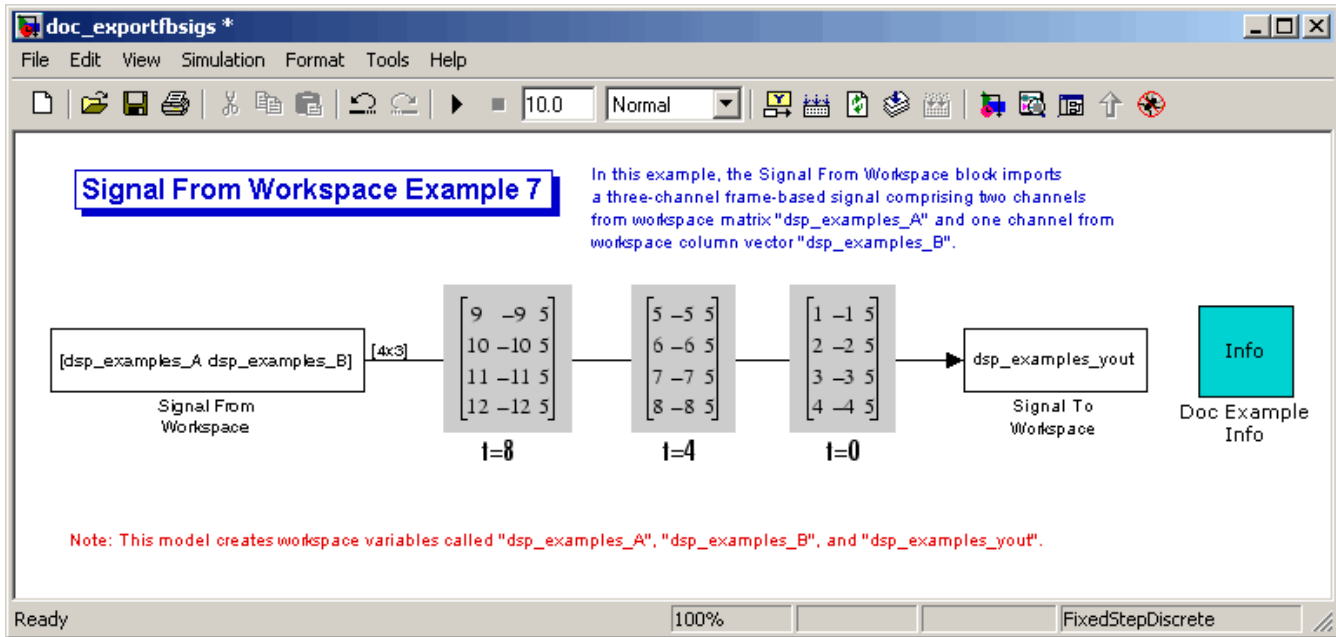
3 Double-click the Signal To Workspace block. Set the block parameters as follows, and then click **OK**:

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

Based on these parameters, the Signal To Workspace block exports its frame-based 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.



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
    
```

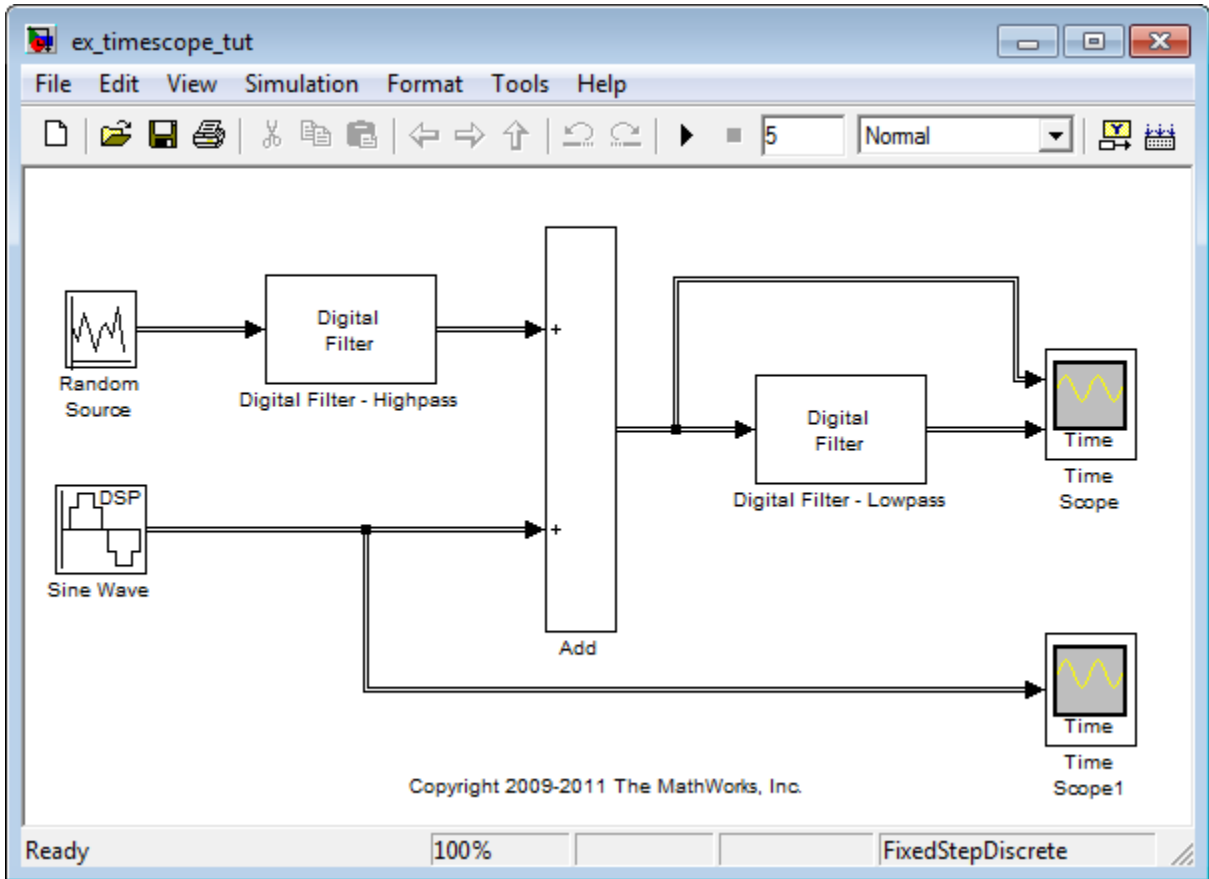
The frames of the signal are concatenated to form a two-dimensional array.

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

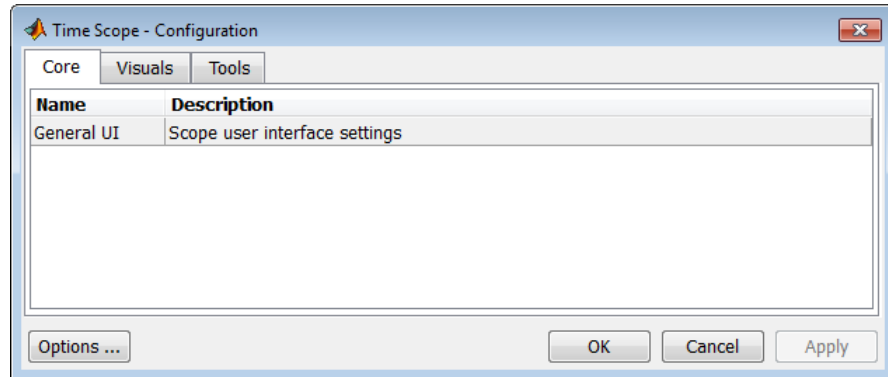
Display Time-Domain Data

In this section...
“Configure the Time Scope” on page 1-75
“Use the Simulation Controls” on page 1-80
“Modify the Time Scope Display” on page 1-82
“Inspect Your Data (Scaling the Axes and Zooming)” on page 1-84
“Manage Multiple Time Scopes” on page 1-86

The following tutorial shows you how to configure the Time Scope blocks in the `ex_timescope_tut` model to display time-domain signals.



The Time Scope – Configuration dialog box provides a central location from which you can change the appearance and behavior of the Time Scope block. To open the Time Scope – Configuration dialog box, double-click the Time Scope block in your model, and select **File > Configuration**.



The Time Scope – Configuration dialog box has three different tabs, **Core**, **Visuals**, and **Tools**, 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 reference page.

Use the following workflow to configure the Time Scope blocks in the `ex_timescope_tut` model:


- 1 “Configure the Time Scope” on page 1-75
- 2 “Use the Simulation Controls” on page 1-80
- 3 “Modify the Time Scope Display” on page 1-82
- 4 “Inspect Your Data (Scaling the Axes and Zooming)” on page 1-84
- 5 “Manage Multiple Time Scopes” on page 1-86

To get started with this tutorial, open the model by typing `ex_timescope_tut` at the MATLAB command line.

Configure the Time Scope

To open the Time Scope – Configuration 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 **File > Configuration**.

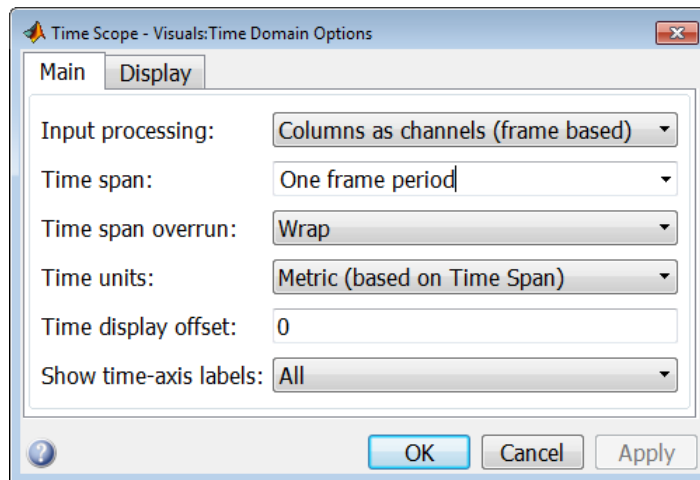
First, you configure the appearance of the Time Scope window and specify how the Time Scope block should interpret input signals. To do so, use the Time Scope – Visuals:Time Domain Options dialog box.

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

To configure the appearance of the Time Scope window and specify how the Time Scope block interprets input signals, in the Time Scope menu, select **View > Properties**. Alternatively, you can do the following:

- 1 In the main Time Scope – Configuration dialog box, click the **Visuals** tab.
- 2 Select **Time Domain**, and click **Options**.

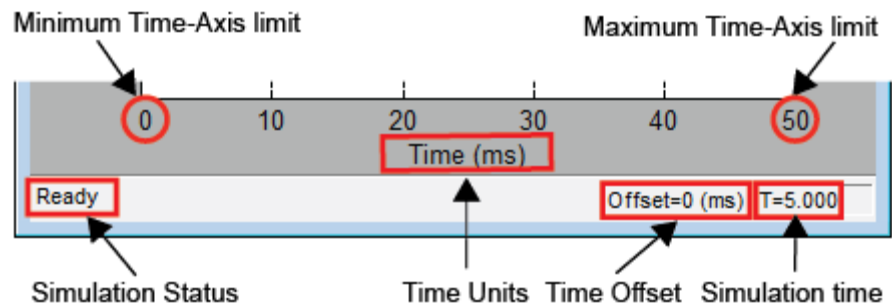


When the Visuals:Time Domain Options dialog box opens, choose the appropriate parameter settings for the **Main** tab, as shown in the following table.

Parameter	Setting
Input processing	Columns as channels (frame based)
Time span	One frame period
Time span overrun	Wrap
Time units	Metric (based on Time Span)
Time display offset	0
Show time-axis labels	All

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).

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. 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.



The value of the minimum Time-axis limit is equal to the **Time display offset** parameter. 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). Click **OK** to save your changes and close the Visuals:Time Domain Options dialog box.

Set Display Properties

In the Visuals:Time Domain Options dialog box, click the **Display** tab. Set the parameters to the values shown in the following table.

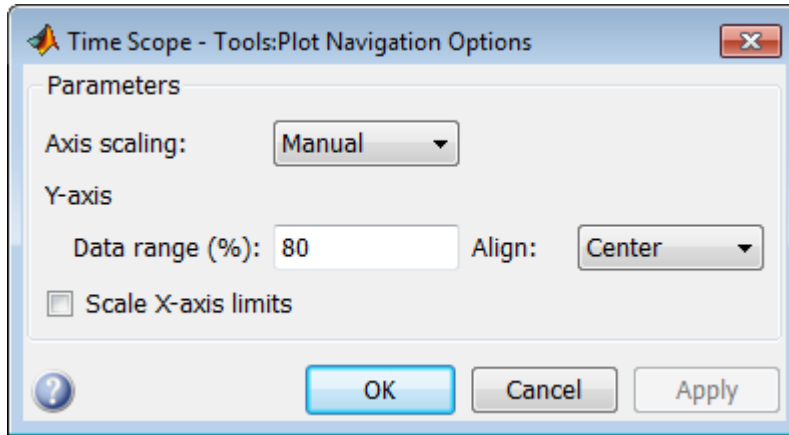
Parameter	Setting
Select display	1
Title	
Show legend	Checked
Show grid	Checked
Plot signal(s) as magnitude and phase	Unchecked
Minimum Y-limit	-2.5
Maximum Y-limit	2.5
Y-axis label	Amplitude

Configure Axis Scaling and Data Alignment

The **Plot Navigation** options for the Time Scope block allow you to control when and how the block scales the axes. These options also control how the block aligns your data with respect to the axes. The following table describes these options.

Parameter	Description
Axis scaling	Allows you to specify when the block should scale the axes. You can choose to scale the axes manually, allow the Time Scope to automatically scale the axes when simulation stops, or allow scaling as needed throughout simulation.
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 <i>y</i> - and <i>x</i> -axis. The higher the value you enter for the <i>y</i> -axis Data range (%) , the tighter the <i>y</i> -axis range is with respect to the minimum and maximum values in your signal. For example, to have your signal cover the entire <i>y</i> -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 <i>y</i> -axis. Additionally, if you select the Scale X-axis limits check box, you can choose to have your data aligned with the right, left, or center of the <i>x</i> -axis.

- 1 To view the Plot Navigation options dialog box, select **Tools > Axes Scaling Options**. Alternatively, in the main Time Scope – Configuration dialog box, click the **Tools** tab, and then click **Options**.



- 2 Set the parameters for the Plot Navigation options dialog box as shown in the following table.

Parameter	Setting
Axis scaling	Manual
Data range (%)	80
Align	Center
Scale X-axis limits	Unchecked


- 3 Click **OK** to save your changes and close the dialog box.

Note If you have not already done so, repeat all of these procedures for the Time Scope1 block 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 single-steps forward 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 start 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 simulate one step button ()

Next, try using keyboard shortcuts to achieve the same result. Press the **Right arrow** key to advance the simulation by a single time step. Alternatively, you can also use the **Page Down** key for this purpose.

- 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 simulation 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 **View** menu. Among other capabilities, this menu allows you to:

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

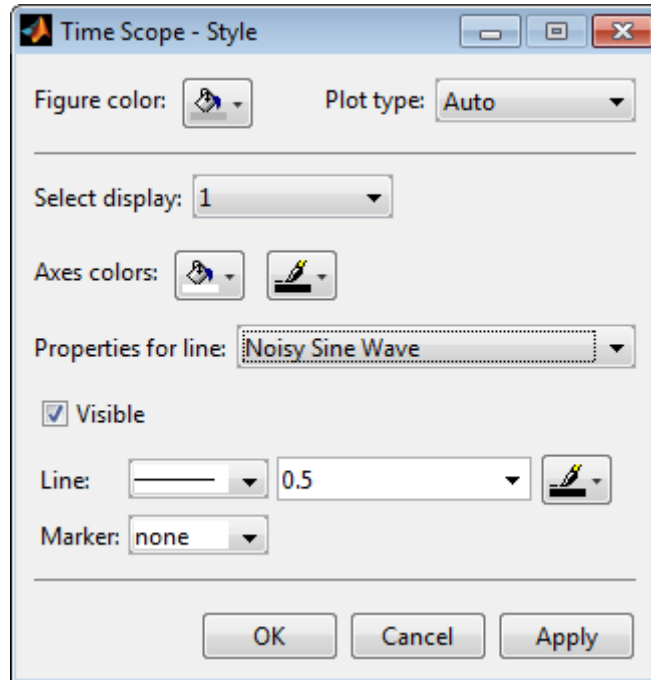
Change Signal Names in the Legend

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 Line Properties

Modify the line properties for the signals in your model using the **View > Style** menu option on the Time Scope window.



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 according to the values shown in the following table.

Block Name	Signal Name	Line	Marker	Color
Time Scope	Noisy Sine Wave	————	none	Black
Time Scope	Filtered Noisy Sine Wave	————	◇	Red
Time Scope1	Original Sine Wave	————	*	Blue

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 > Simulation Toolbar**. Doing so removes the simulation toolbar from the Time Scope window and also removes the check mark from next to the **Simulation Toolbar** option in the **View** menu.
- You can choose to show the simulation toolbar again at any time by selecting **View > Simulation Toolbar**.

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

Inspect Your Data (Scaling the Axes and Zooming)

The Time Scope block 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**.

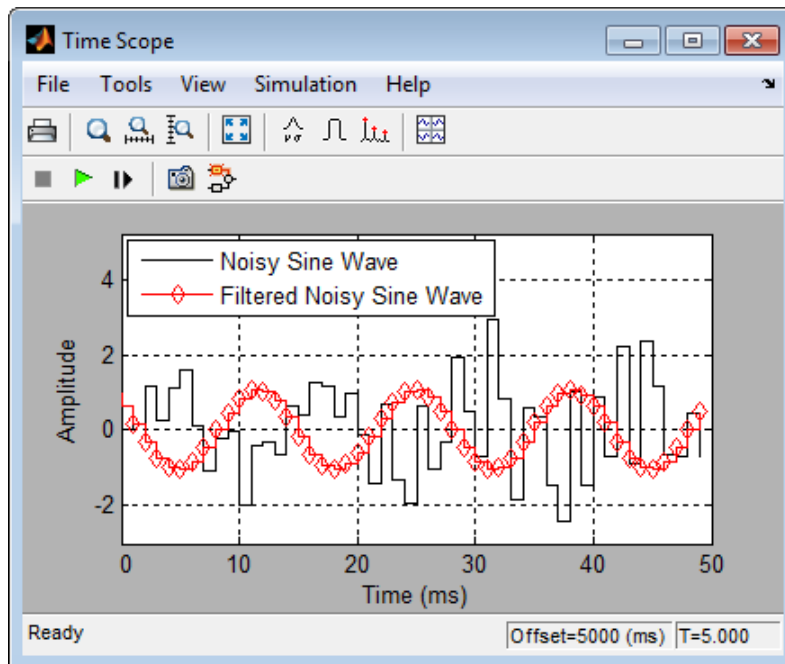
Adjust White Space Around the Signal

You can control how much white space surrounds your signal and where your signal appears in relation to the axes. To adjust the amount of white space surrounding your signal and realign it with the axes, you must first open the Time Scope – Tools:Plot Navigation Options dialog box. From the Time Scope menu, select **Tools > Axes Scaling Options** .

In the Time Scope – 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 increase the white space surrounding your signal, set the **Data range (%)** parameter on the **Time Scope – Tools:Plot Navigation Options** dialog box to 50.
- 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.
- 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

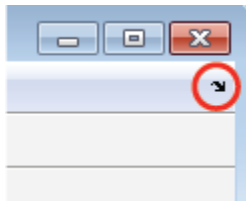
Sometimes, you have multiple Time Scope blocks in your model and need to find the location of one that corresponds to the active Time Scope window. In such cases, you can use the **View > Highlight Simulink Block** menu

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.

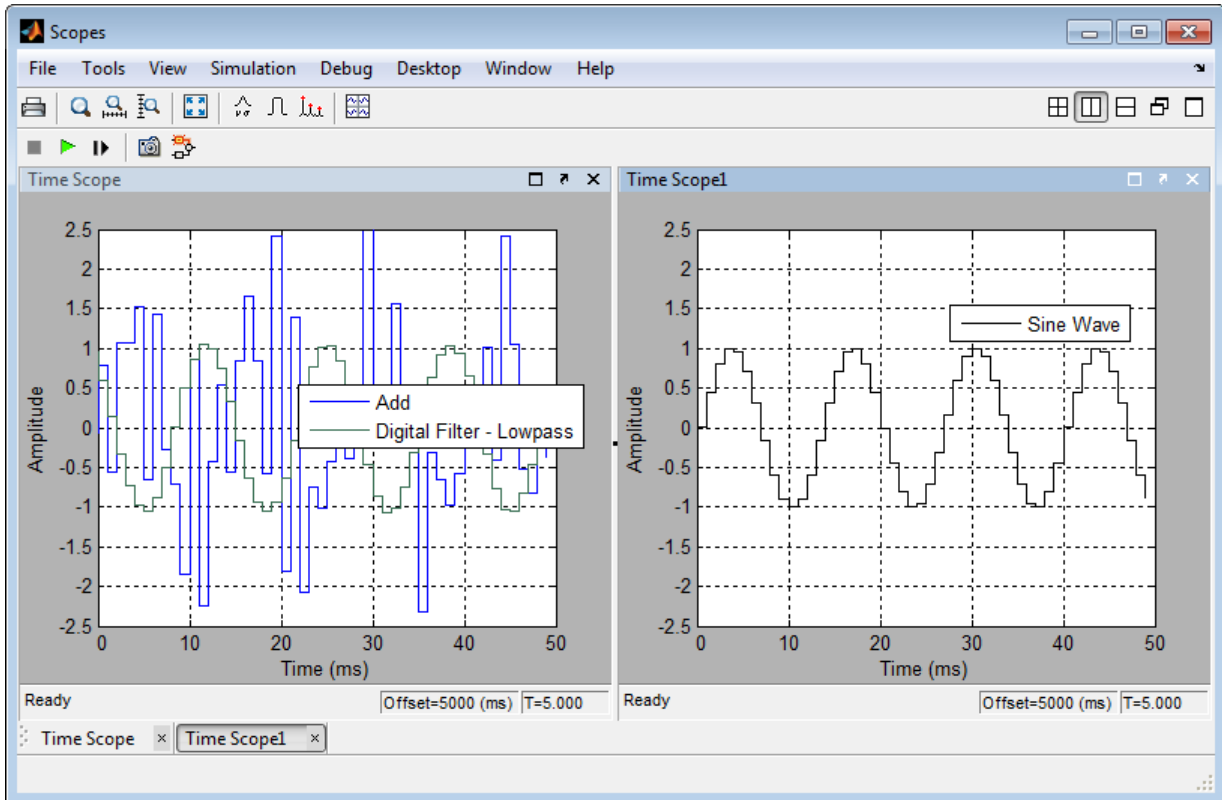
Docking Time Scope Windows in the Scopes Group Container

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 Undock Scopes button () at the top-right corner of the container. 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 Time Scope button () in the title bar of each docked instance. If you

close the Scopes group container, all docked instances of Time Scope close. The Simulink model continues to run.

For more information on docking figures, see *Docking Figures in the Desktop* in the MATLAB documentation.

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**.

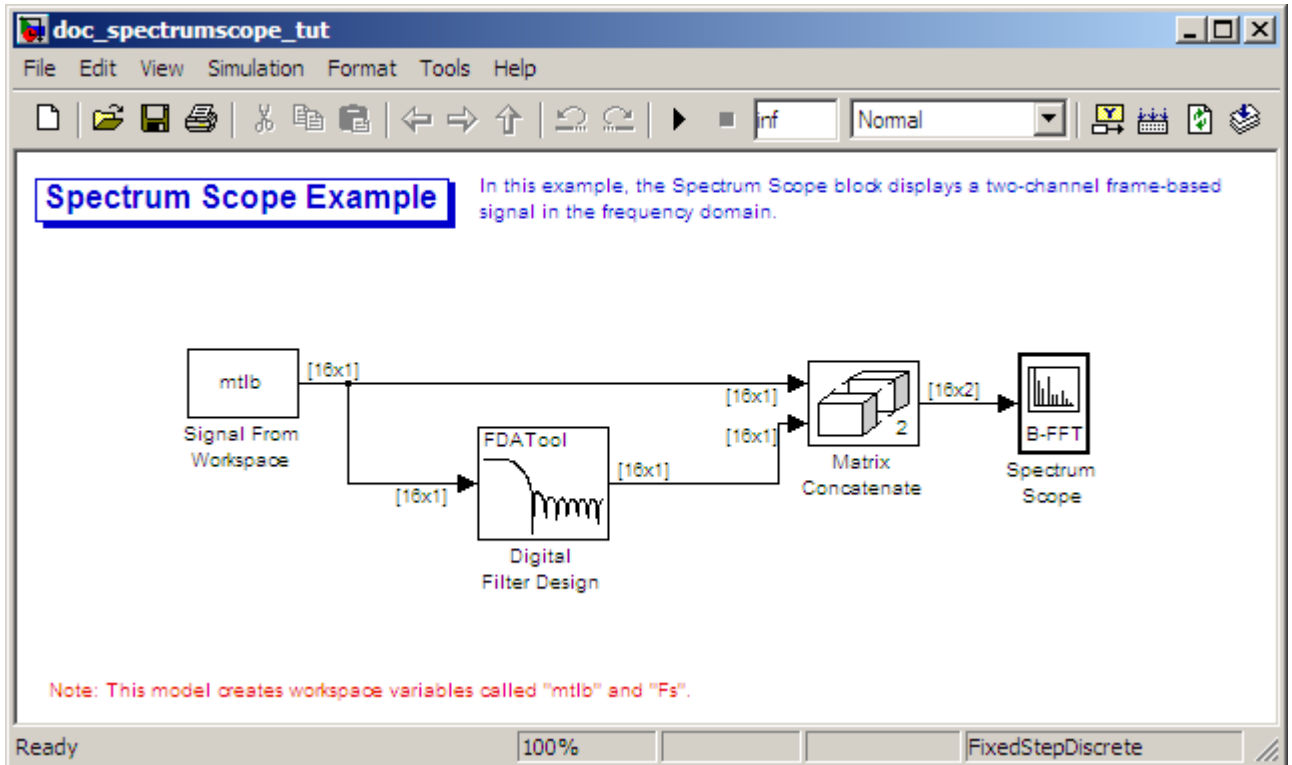
Display Frequency-Domain Data

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 Scope, Matrix Viewer, and Waterfall Scope blocks.

You can use the Spectrum Scope block to display the frequency spectra of time-domain input data. In contrast to the Vector Scope block, the Spectrum Scope block computes the FFT of the input signal internally, transforming it into the frequency domain. In this example, you use a Spectrum Scope block to display the frequency content of two frame-based signals simultaneously:

1 At the MATLAB command prompt, type `ex_spectrumscope_tut`.

The Spectrum Scope Example opens.



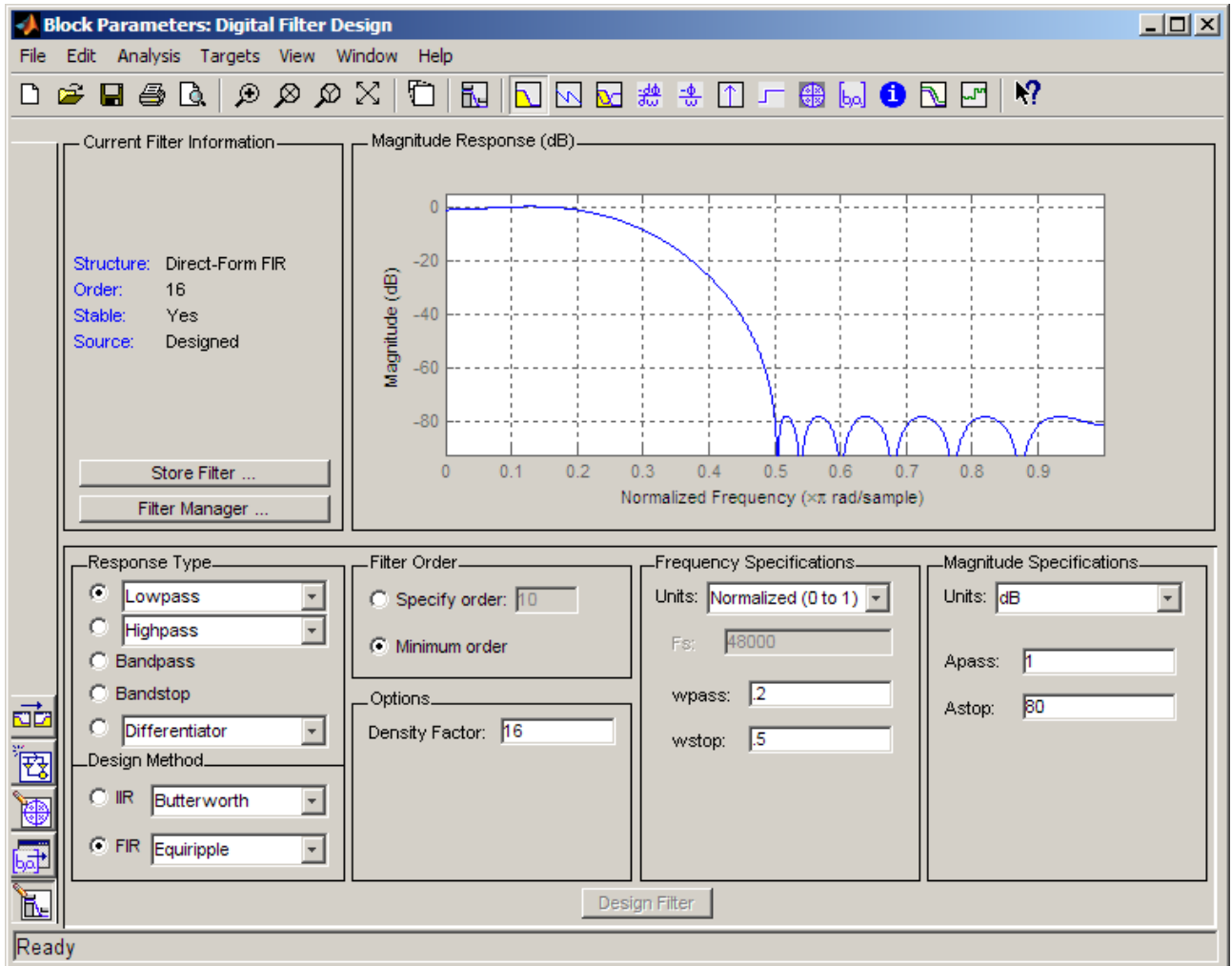
Also, the variables `Fs` and `mtlb` are loaded into the MATLAB workspace.

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.

- Use the Digital Filter Design block to filter the input signal to produce two distinct signals to send to the Spectrum Scope block. Use the default parameters.



- 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

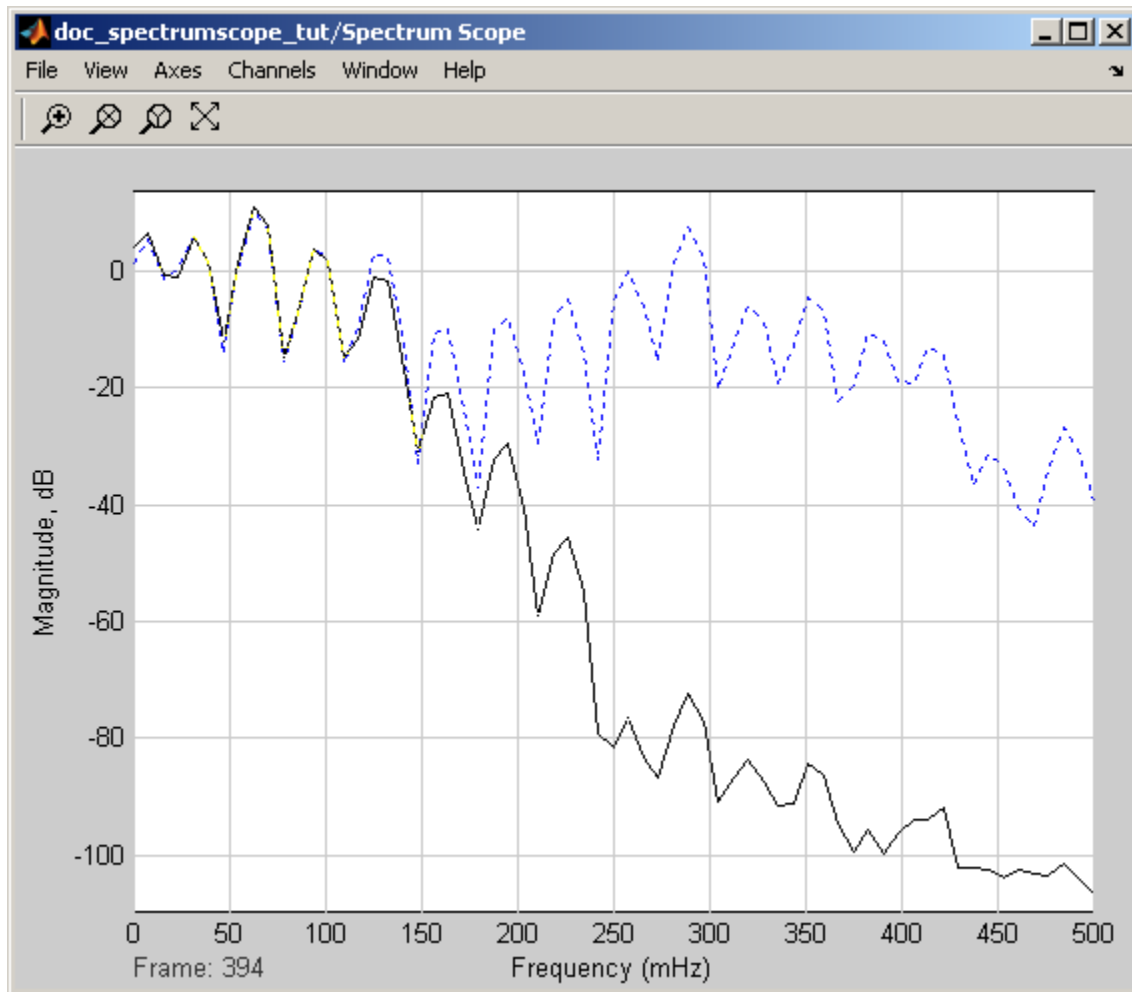
The Matrix Concatenate block combines the two signals so that each column corresponds to a different signal.

- 5 Double-click the Spectrum Scope block. On the **Scope Properties** tab, set the block parameters as follows, and then click **OK**:
 - Select the **Buffer input** check box.
 - **Buffer size** = 128
 - **Buffer overlap** = 64
 - **Window type** = Hann
 - **Window sampling** = Periodic
 - Clear the **Specify FFT length** check box.
 - **Number of spectral averages** = 2

Based on these parameters, the Spectrum Scope block buffers each input channel to a new frame size of 128 (from the original frame size of 16) with an overlap of 64 samples between consecutive frames. Because **Specify FFT length** is not selected, the frame size of 128 is used as the number of frequency points in the FFT. This is the number of points plotted for each channel every time the scope display is updated.

- 6 Run the model.
- 7 While the model is running, right-click in the Spectrum Scope window. Point to **Ch1**, point to **Style**, and point to **:**. Right-click again and point to **Autoscale**.

The Spectrum Scope block computes the FFT of each of the input signals. It then displays the magnitude of the frequency-domain signals in the Spectrum Scope window.



The FFT of the first input signal, from column one, is the blue dotted line. The FFT of the second input signal, from column two, is the black solid line. Every time the scope display is updated, 128 points are plotted for each channel.

You have now used the Spectrum Time Scope block to display two, frame-based signals in the frequency domain.

Data and Signal Management

Learn concepts such as sample- and frame-based processing, sample rate, delay and latency.

- “Sample- and Frame-Based Concepts” on page 2-2
- “Inspect Sample Rates and Frame Rates in Simulink” on page 2-8
- “Convert Sample and Frame Rates in Simulink” on page 2-17
- “Convert Frame Status” on page 2-39
- “Delay and Latency” on page 2-55

Sample- and Frame-Based Concepts

In this section...
“Sample- and Frame-Based Signals” on page 2-2
“Model Sample- and Frame-Based Signals in MATLAB and Simulink” on page 2-3
“What Is Sample-Based Processing?” on page 2-4
“What Is Frame-Based Processing?” on page 2-5

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 2-6 for more information.

DSP System Toolbox Source Blocks	Create Sample-Based Signals	Create Frame-Based Signals
Chirp	X	X
Constant	X	X
Constant Diagonal Matrix	X	
Discrete Impulse	X	X
DSP Constant (Obsolete)	X	
From Audio Device	X	X
From Multimedia File	X	X
Identity Matrix	X	
MIDI Controls	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. Alternatively, when you are using System objects in MATLAB, you specify the processing mode using the `FrameBasedProcessing` property. The following table shows the common parameter settings you can use to perform sample- and frame-based processing in MATLAB and Simulink.

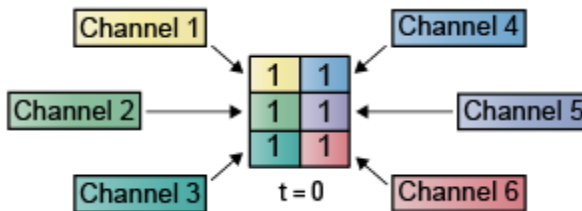
	Sample-Based Processing	Frame-Based Processing
MATLAB — System objects	FrameBasedProcessing = False	FrameBasedProcessing = True
Simulink — Blocks	Input processing = Elements as channels (sample based)	Input processing = Columns as channels (frame based)

Set the FrameBasedProcessing Property of a System Object

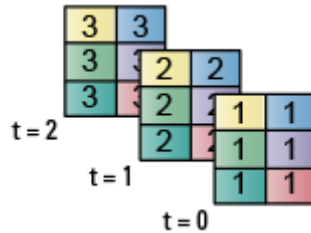
All System objects support sample-based processing and some System objects support both sample- and frame-based processing. To specify how your object should process input data, you set the FrameBasedProcessing property. The property has a default value of true, which enables frame-based processing. To specify sample-based processing, set the FrameBasedProcessing property to false.

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.



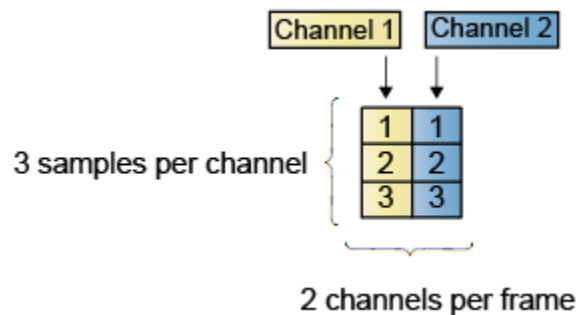
When you configure a block to perform sample-based processing, the block interprets scalar input as a single-channel signal. Similarly, the block interprets an M -by- N matrix as multichannel signal with $M \times N$ independent channels. For example, in sample-based processing, blocks interpret the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



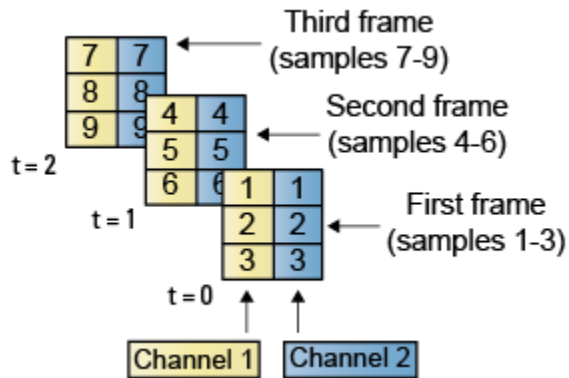
For more information about the recent changes to frame-based processing, see the “Frame-Based Processing” section of the *DSP System Toolbox Release Notes*.

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.



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” 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.

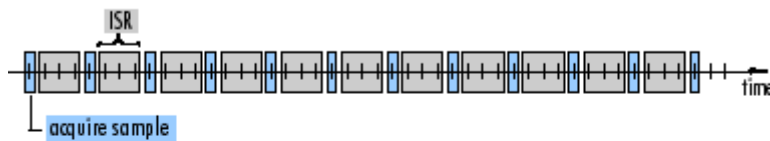
Accelerate Real-Time Systems. Frame-based data is a common format in real-time systems. Data acquisition hardware often operates by accumulating a large number of signal samples at a high rate, and then propagating those samples to the real-time system as a block of data. This type of propagation maximizes 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

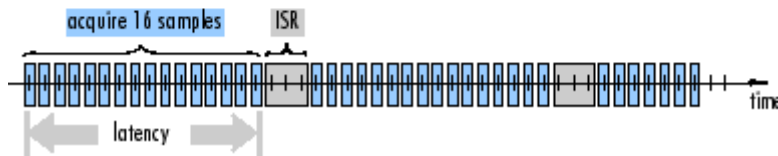
acquisition of a sample. The thicker blocks each represent the time elapsed during the interrupt service routine (ISR) that reads the data from the hardware.

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 2-55.

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.

Inspect Sample Rates and Frame Rates in Simulink

Note Starting in R2010b, many DSP System Toolbox blocks received a new parameter to control whether they perform sample- or frame-based processing. The following content has not been updated to reflect this change. For more information, see the “Frame-Based Processing” section of the Release Notes.

In this section...
“Sample Rate and Frame Rate Concepts” on page 2-8
“Inspect Sample-Based Signals Using the Probe Block” on page 2-10
“Inspect Frame-Based Signals Using the Probe Block” on page 2-11
“Inspect Sample-Based Signals Using Color Coding” on page 2-13
“Inspect Frame-Based Signals Using Color Coding” on page 2-15

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-based 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-based 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-based signal is the quotient of the frame period and the frame size, M :

$$T_s = T_f / M$$

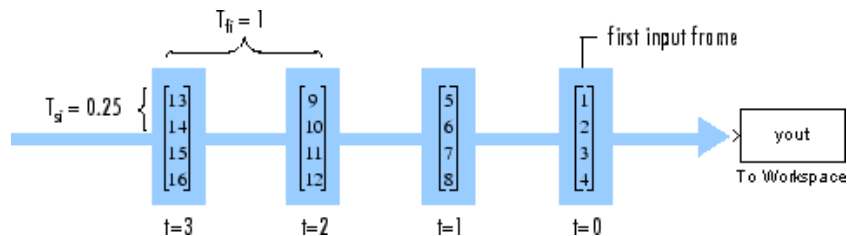
More specifically, the sample periods of inputs (T_{si}) and outputs (T_{so}) are related to their respective frame periods by

$$T_{si} = T_{fi} / M_i$$

$$T_{so} = T_{fo} / M_o$$

where M_i and M_o are the input and output frame sizes, respectively.

The illustration below shows a single-channel, frame-based 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.

Inspect Sample-Based Signals Using the Probe Block

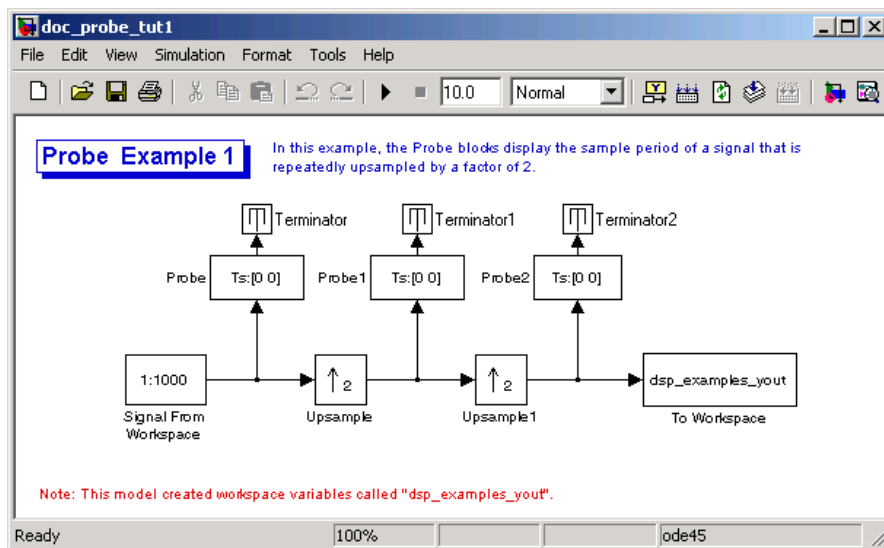
You can use the Probe block to display the sample period of a sample-based signal. For sample-based signals, the Probe block displays the label T_s , the sample period of the sequence, followed by a two-element vector. The left element is the period of the signal being measured. The right element is 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.

In this example, you use the Probe block to display the sample period of a sample-based signal:

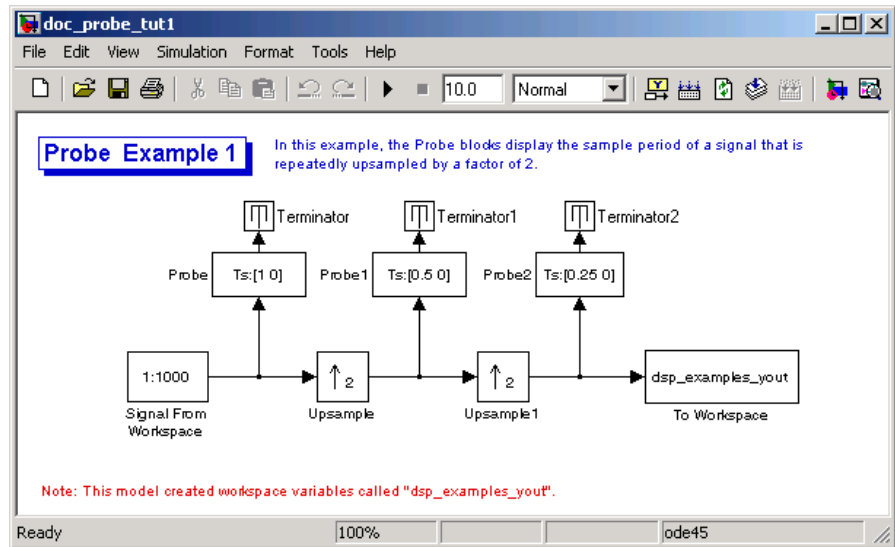
- 1 At the MATLAB command prompt, type `ex_probe_tut1`.

The Probe Example 1 model opens.



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 sample-based 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.

Inspect Frame-Based Signals Using the Probe Block

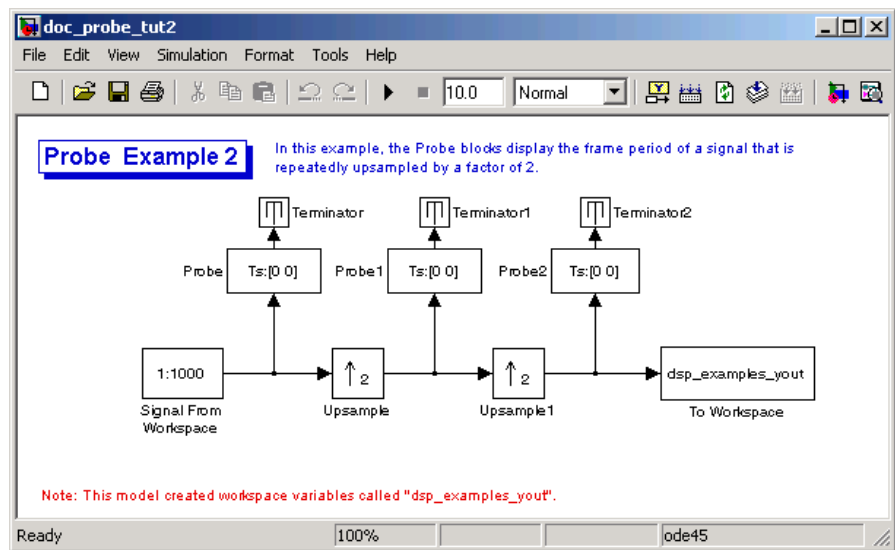
You can use the Probe block to display the frame period of a frame-based signal. For frame-based signals, the block displays the label Tf, the frame period of the sequence, followed by a two-element vector. The left element is the period of the signal being measured. The right element is the signal's sample time offset, which is usually 0.

Note Simulink offers the ability to shift a signal's sample times 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.

In this example, you use the Probe block to display the frame period of a frame-based signal:

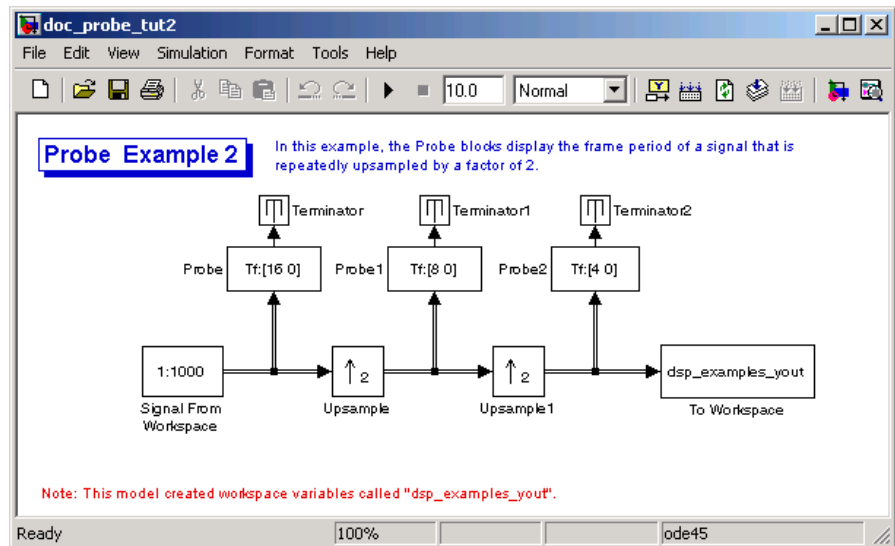
- 1 At the MATLAB command prompt, type `ex_probe_tut2`.

The Probe Example 2 model opens.



- 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 is a frame-based signal with 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 sample period of 4 seconds.

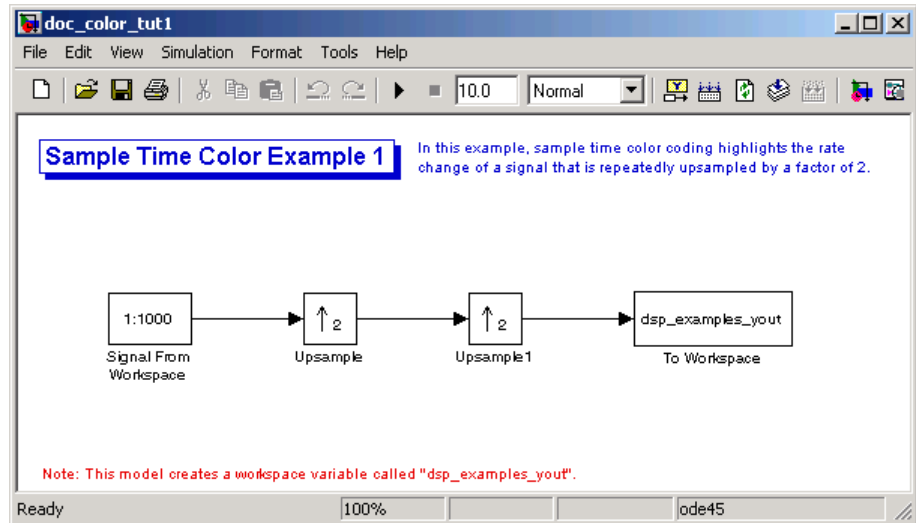
Note that the sample rate conversion is implemented through a change in the frame period rather than the frame size.

Inspect Sample-Based Signals Using Color Coding

In the following example, you use sample time color coding to view the sample rate of a sample-based signal:

- 1 At the MATLAB command prompt, type `ex_color_tut1`.

The Sample Time Color Example 1 model opens.

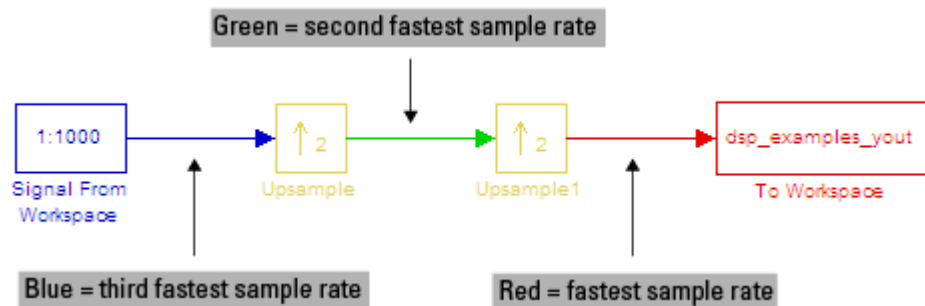


- 2 From the **Format** menu, point to **Sample Time Display**, and select **Colors**.

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 sample-based signal in this model has a different sample rate. Therefore, each signal is assigned a different color.

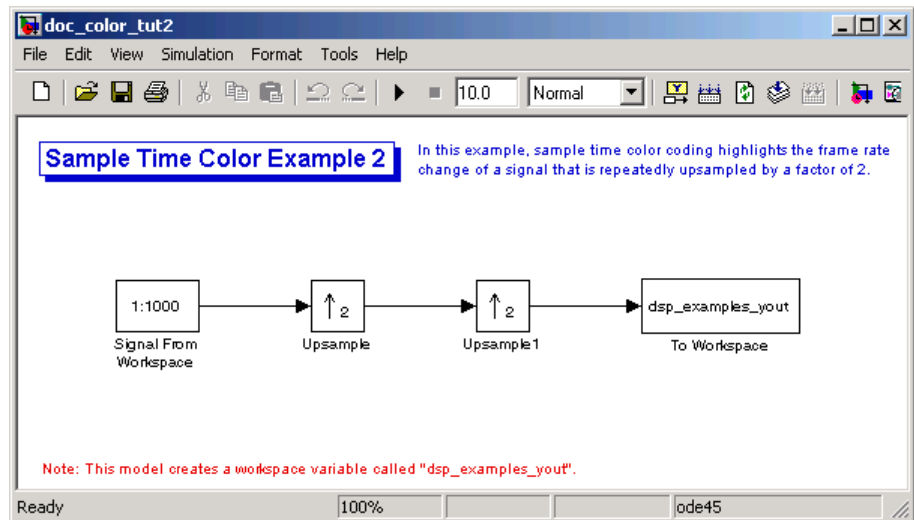
For more information about sample time color coding, see “How to View Sample Time Information” in the Simulink documentation.

Inspect Frame-Based Signals Using Color Coding

In this example, you use sample time color coding to view the frame rate of a frame-based signal:

- 1 At the MATLAB command prompt, type `ex_color_tut2`.

The Sample Time Color Example 2 model opens.

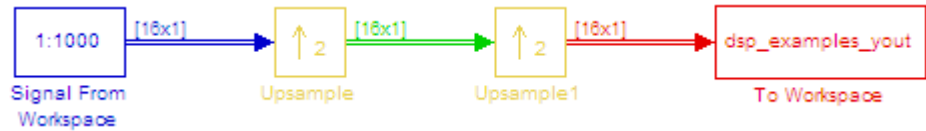


- 2 To turn on sample time color coding, from the **Format** menu, point to **Sample Time Display**, and select **Colors**.

Simulink now assigns each frame rate a different color.

- 3 Run the model.

The model should now look similar to the following figure:

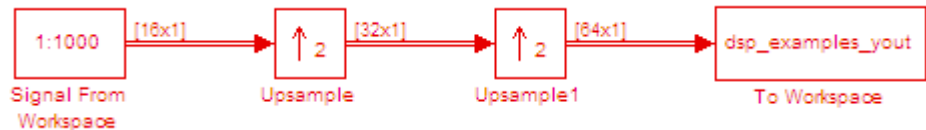


Because the **Rate options** parameter in the Upsample blocks is set to Allow multirate processing, each Upsample block changes the frame rate. Therefore, each frame-based signal in the model is assigned a different color.

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 “Displaying Sample Time Colors” in the Simulink documentation.

Convert Sample and Frame Rates in Simulink

Note Starting in R2010b, many DSP System Toolbox blocks received a new parameter to control whether they perform sample- or frame-based processing. The following content has not been updated to reflect this change. For more information, see the “Frame-Based Processing” section of the Release Notes.

In this section...

“Rate Conversion Blocks” on page 2-17

“Rate Conversion by Frame-Rate Adjustment” on page 2-18

“Rate Conversion by Frame-Size Adjustment” on page 2-21

“Avoid Unintended Rate Conversion” on page 2-24

“Frame Rebuffering Blocks” on page 2-30

“Buffer Signals by Preserving the Sample Period” on page 2-33

“Buffer Signals by Altering the Sample Period” on page 2-36

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 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

Block	Library
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.

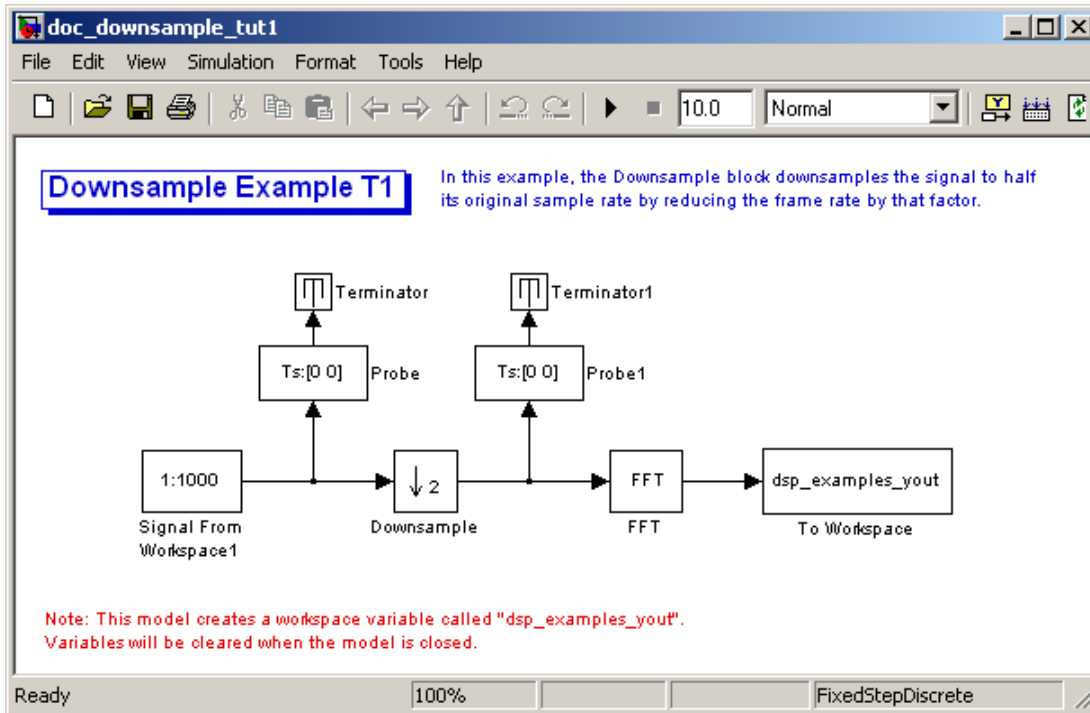
Note When a Simulink model contains signals with various frame rates, the model is called *multirate*. You can find a discussion of multirate models in “Excess Algorithmic Delay (Tasking Latency)” on page 2-63. Also see “Scheduling” in the Simulink Coder™ documentation.

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 ($M_o = M_f$). 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 **Format** menu, point to **Port/Signal Displays**, 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 **Block Parameters: Signal From Workspace** dialog box opens.

- 4 Set the block parameters as follows:

- **Sample time** = 0.125
- **Samples per frame** = 8

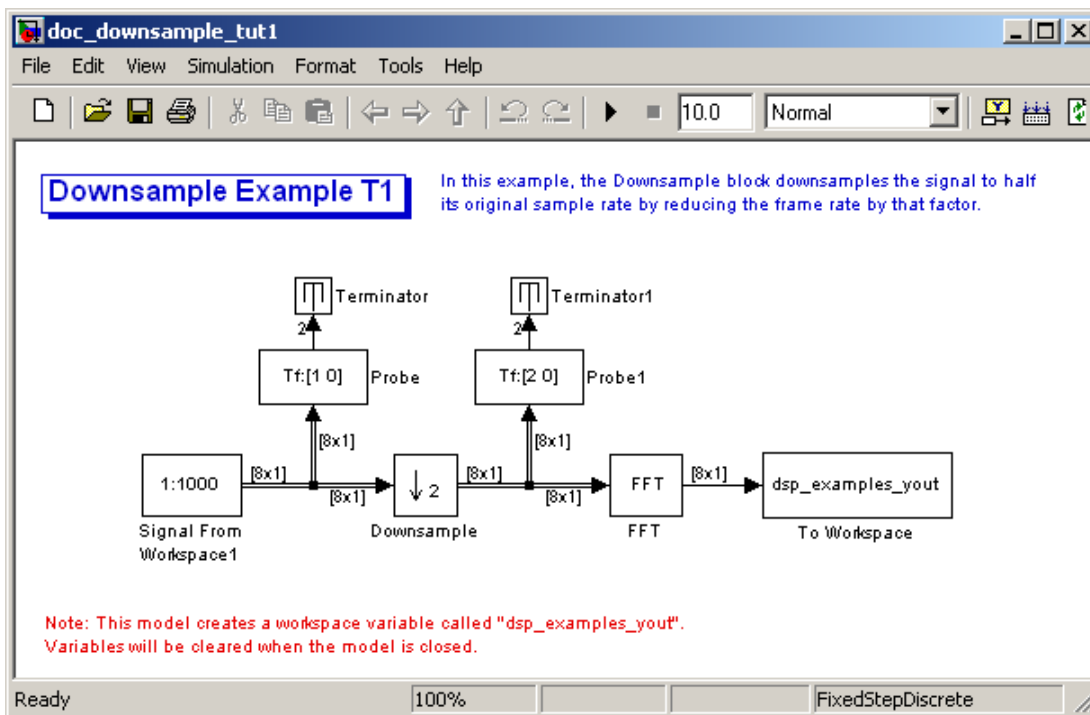
Based on these parameters, the Signal From Workspace block outputs a frame-based signal with a sample period of 0.125 second and a frame size of 8.

- 5 Save these parameters and close the dialog box by clicking **OK**.
- 6 Double-click the Downsample block. The **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 second.

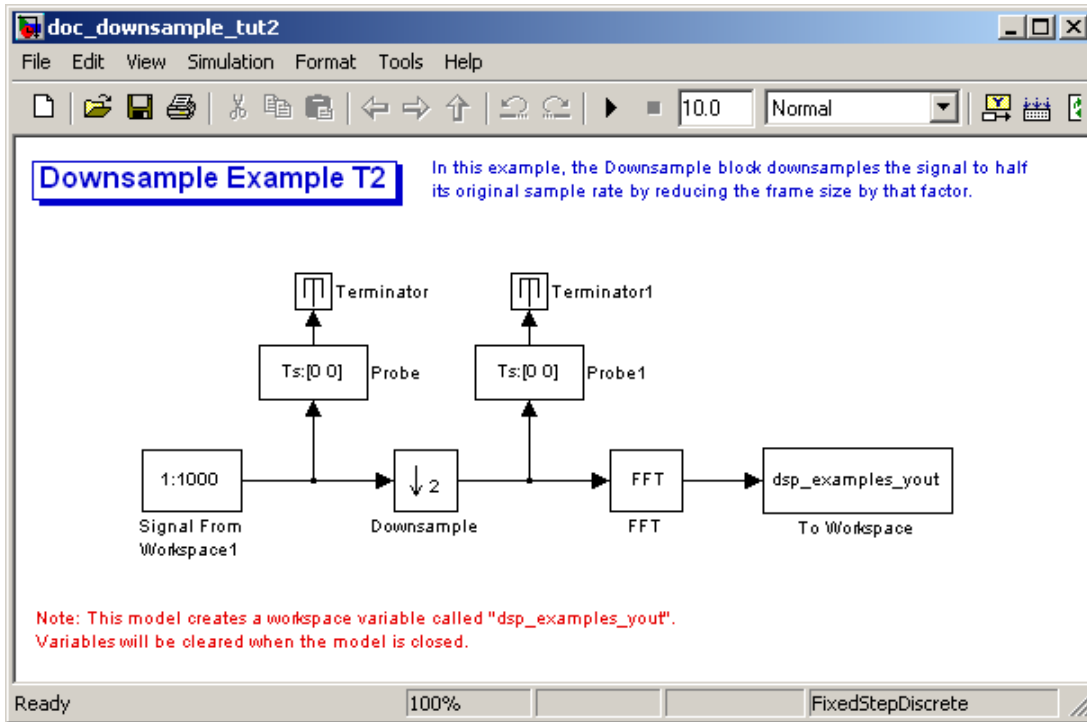
The second Probe block in the model verifies that the output from the Downsample block has a frame period, T_{fo} , of 2 seconds, twice the frame period of the input. However, because the frame rate of the output, $1/T_{fo}$, is 0.5 second, the Downsample block actually downsampled the original signal 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 **Format** menu, point to **Port/Signal Displays**, and select **Signal Dimensions**.

When you run the model, the dimensions the signals appear next to the lines connecting the blocks.

- 3 Double-click the Signal From Workspace block. The **Block Parameters: Signal From Workspace** dialog box opens.

- 4 Set the block parameters as follows:

- **Sample time** = 0.125
- **Samples per frame** = 8

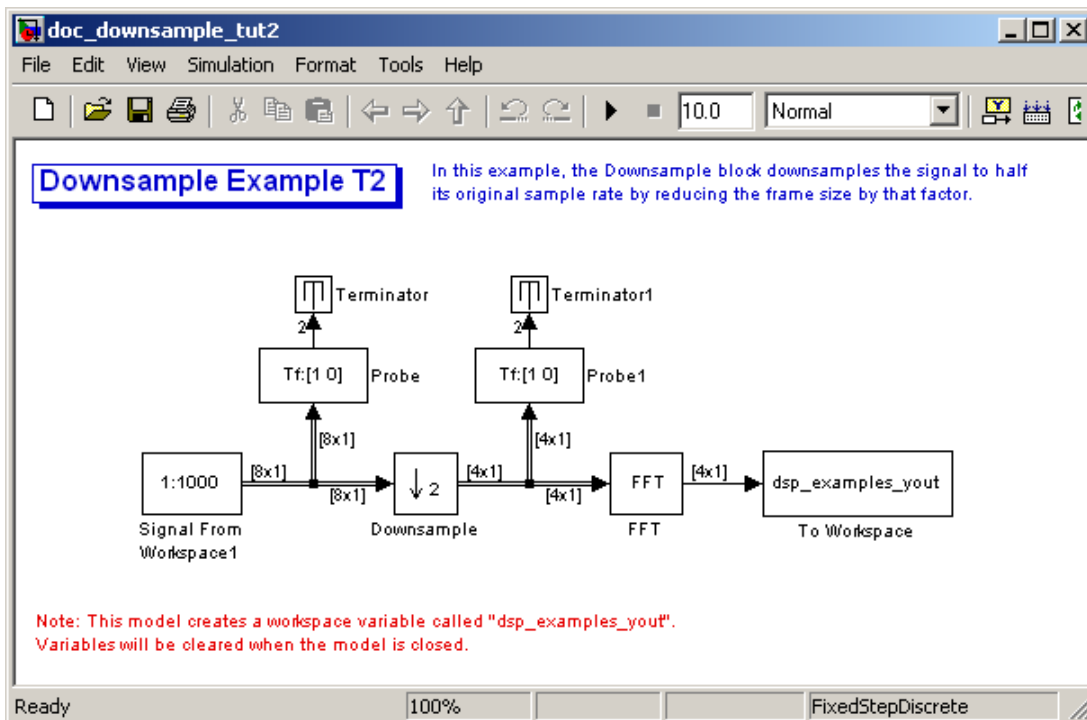
Based on these parameters, the Signal From Workspace block outputs a frame-based signal with a sample period of 0.125 second and a frame size of 8.

- 5 Save these parameters and close the dialog box by clicking **OK**.
- 6 Double-click the Downsample block. The **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 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,

$T_{so} = T_{fo} / M_o$, now has a sample period of 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 **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**

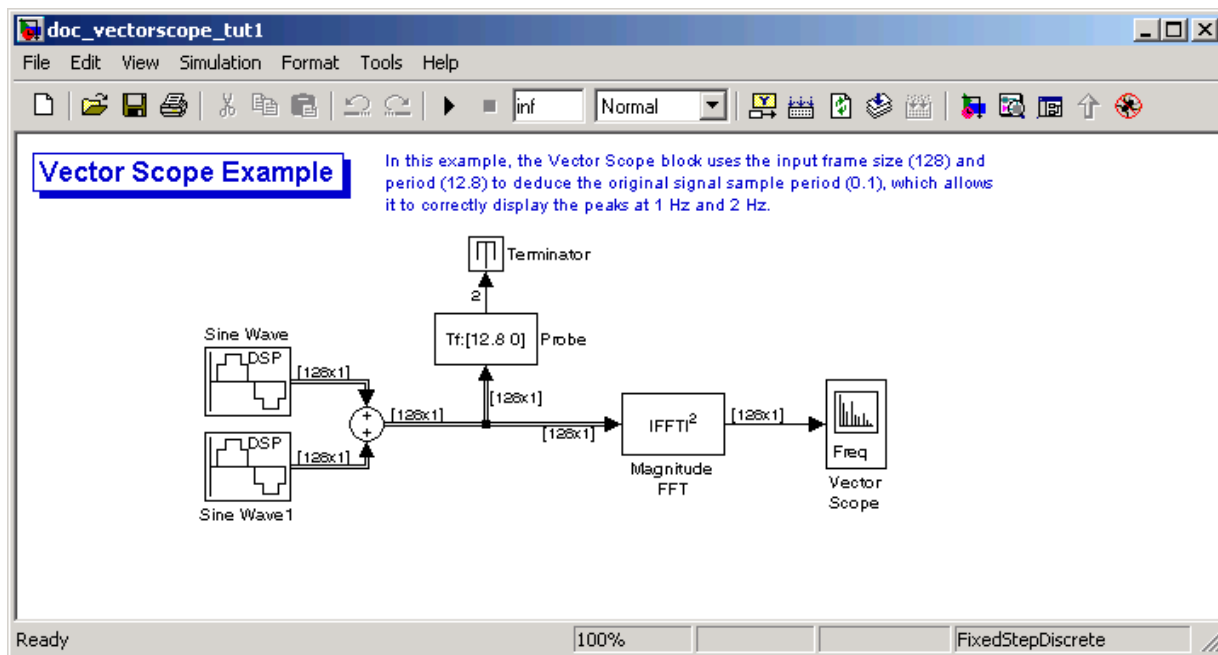
Based on the **Sample time** and the **Samples per frame** parameters, the Sine Wave outputs a sinusoid with a frame period of $128 \cdot 0.1$ or 12.8 seconds.

- 7** Double-click the Magnitude FFT block. The **Block Parameters: Magnitude FFT** dialog box opens.
- 8** Select the **Inherit FFT length from input dimensions** check box, and then click **OK**.

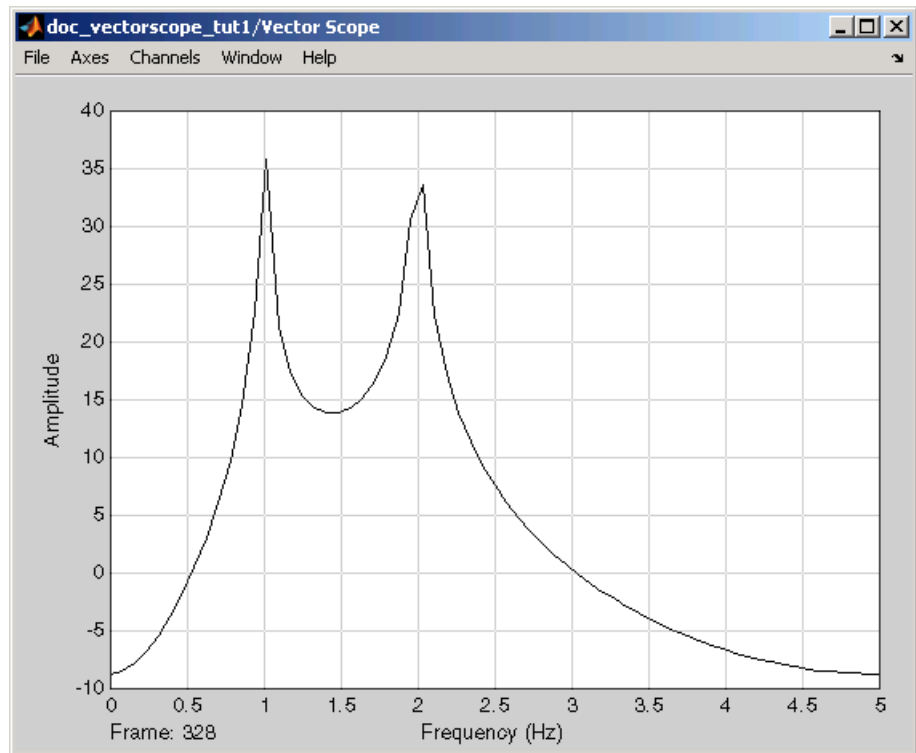
This setting instructs the block to use the input frame size (128) as the FFT length (which is also the output size).

- 9** Double-click the Vector Scope block.
- 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 **Block Parameters: Magnitude FFT** dialog box opens.

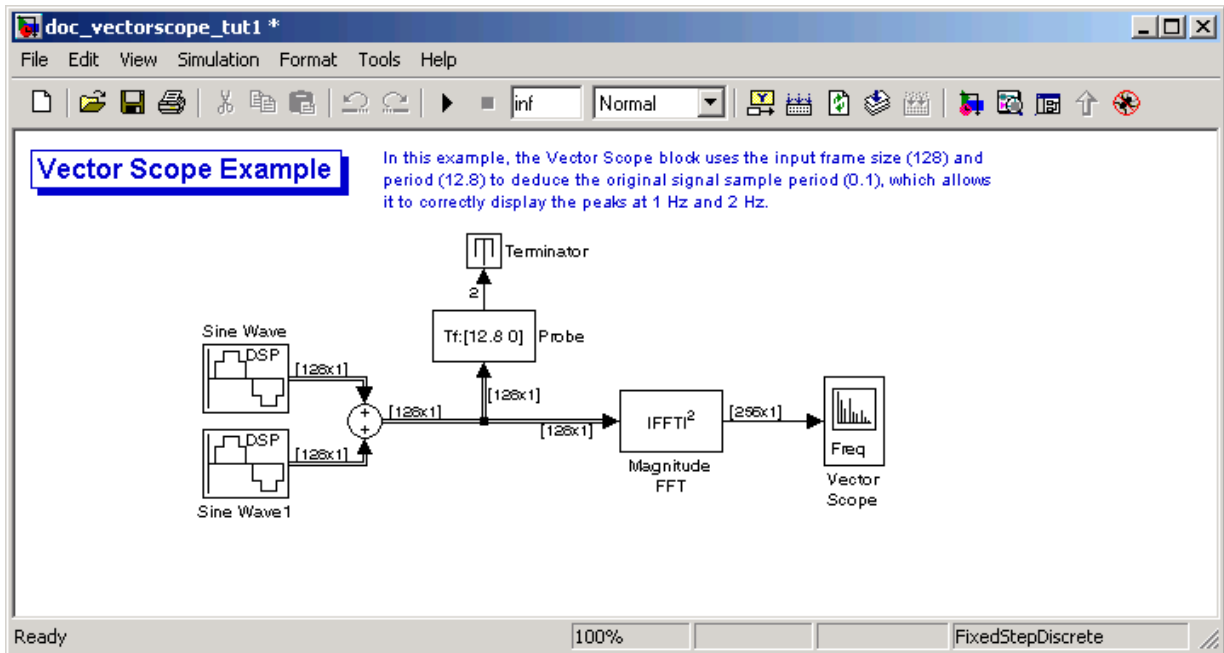
13 Set the block parameters as follows:

- Clear the **Inherit FFT length from input dimensions** check box.
- Set the **FFT length** parameter to 256.

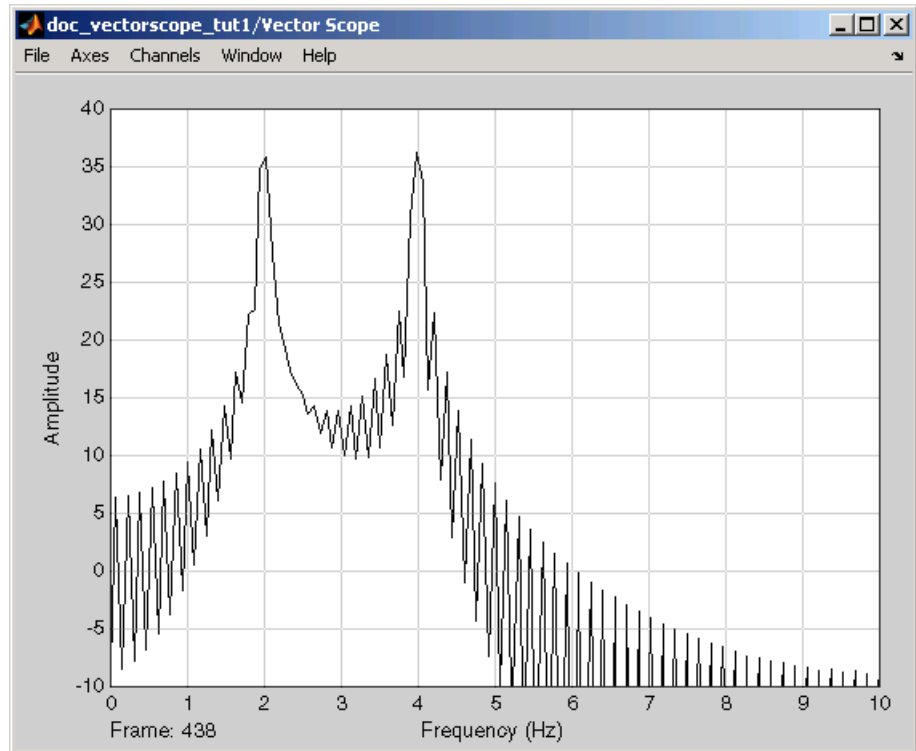
Based on these parameters, the Magnitude FFT block zero-pads the length-128 input frame to a length of 256 before performing the FFT.

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

Block	Library
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 frame-based signal to a sample-based signal (frame size = 1)

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-based signal to its sample-based equivalent, 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-based input.

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.

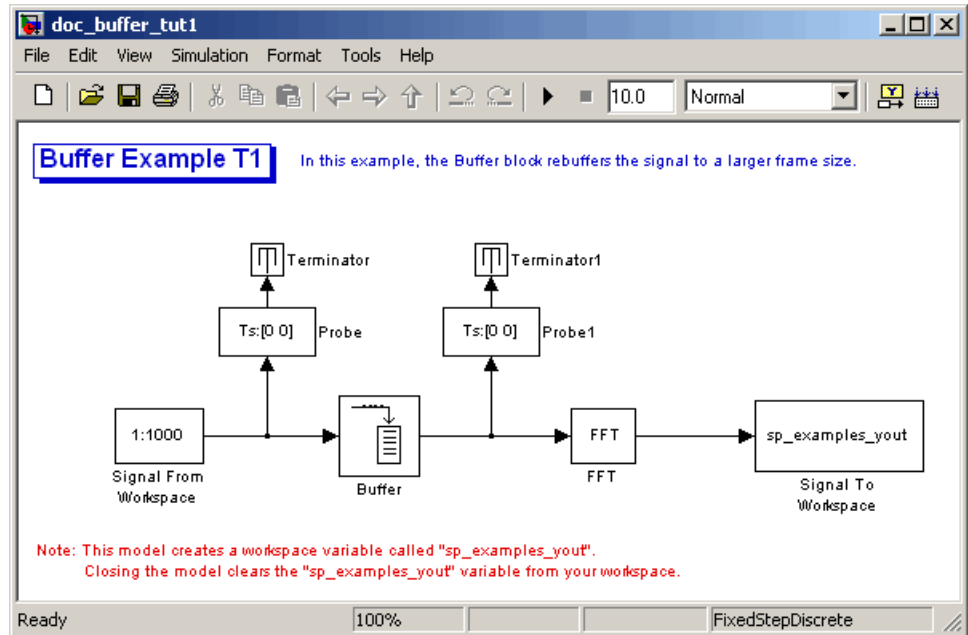
In all of these cases, the sample period of the output sequence is *not* equal to the sample period of the input sequence.

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 **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 frame-based 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 **Block Parameters: Buffer** dialog box opens.

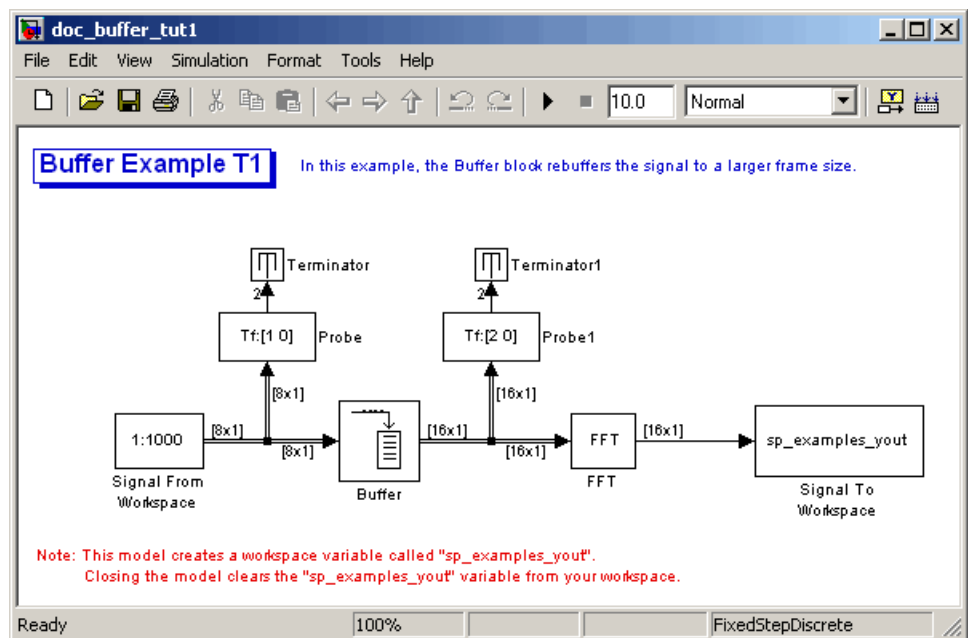
6 Set the parameters as follows, and then click **OK**:

- **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.



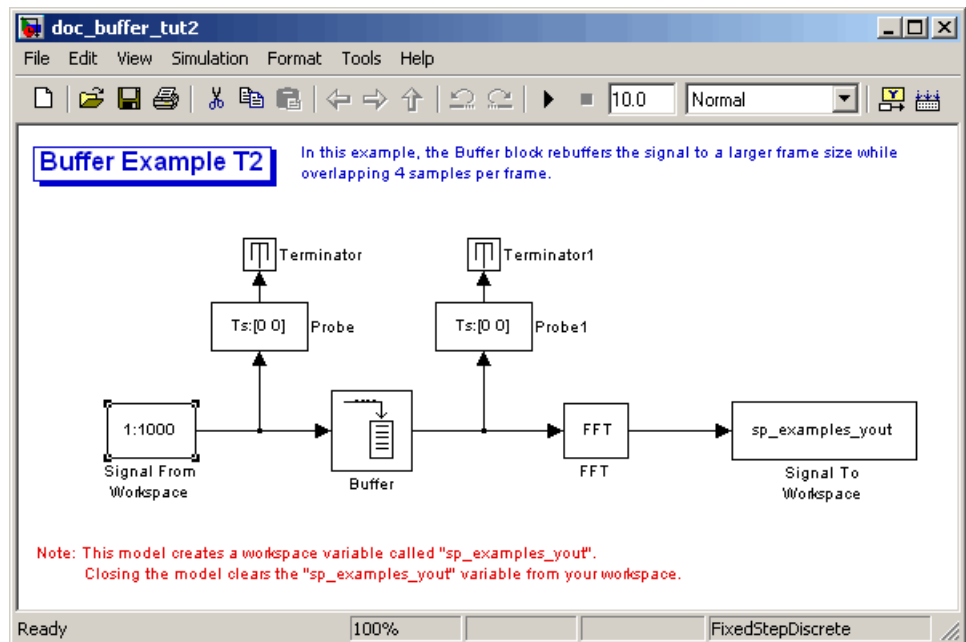
Note that the input to the Buffer block has a frame size of 8 and the output of the block has a frame size of 16. As shown by the Probe blocks, the rebuffering process doubles the frame period from 1 to 2 seconds.

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 **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

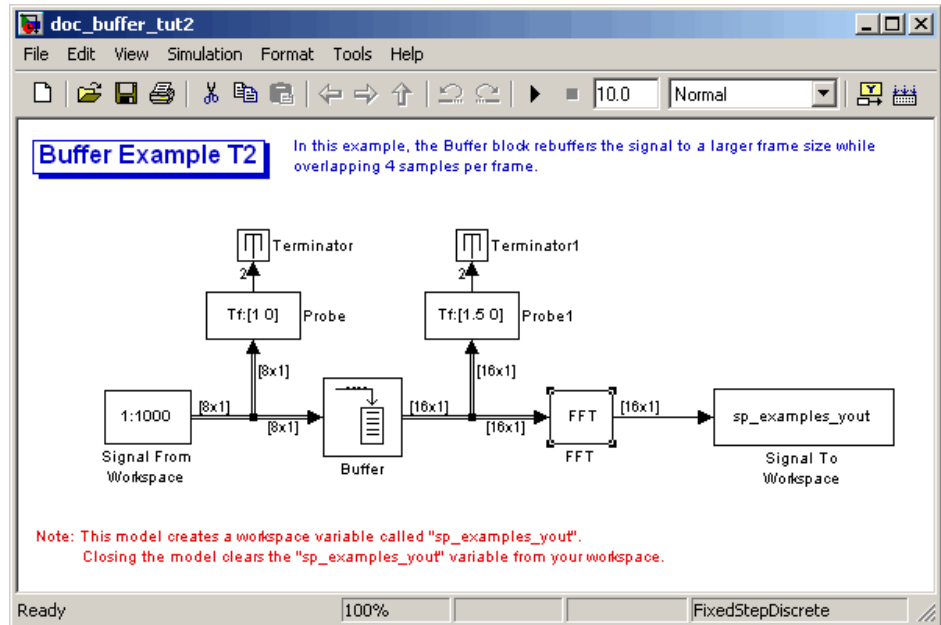
Based on these parameters, the Signal from Workspace block outputs a frame-based signal with a sample period of 0.125 second. Each output frame contains eight samples.

- 4** Save these parameters and close the dialog box by clicking **OK**.
- 5** Double-click the Buffer block. The **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.

Convert Frame Status

Note Starting in R2010b, many DSP System Toolbox blocks received a new parameter to control whether they perform sample- or frame-based processing. The following content has not been updated to reflect this change. For more information, see the “Frame-Based Processing” section of the Release Notes.

In this section...

“Frame Status” on page 2-39

“Buffer Sample-Based Signals into Frame-Based Signals” on page 2-39

“Buffer Sample-Based Signals into Frame-Based Signals with Overlap” on page 2-42

“Buffer Frame-Based Signals into Other Frame-Based Signals” on page 2-47

“Buffer Delay and Initial Conditions” on page 2-50

“Unbuffer Frame-Based Signals into Sample-Based Signals” on page 2-51

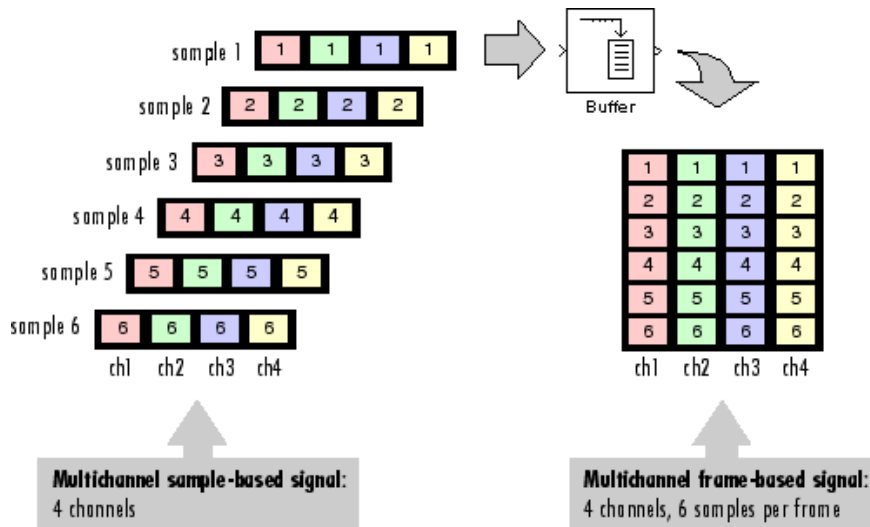
Frame Status

The frame status of a signal refers to whether the signal is sample based or frame based. In a Simulink model, the frame status is symbolized by a single line, \rightarrow , for a sample-based signal and a double line, \Rightarrow for a frame-based signal. One way to convert a sample-based signal to a frame-based signal is by using the Buffer block. You can convert a frame-based signal to a sample-based signal using the Unbuffer block. To change the frame status of a signal without performing a buffering operation, use the Frame Conversion block in the Signal Attributes library.

Buffer Sample-Based Signals into Frame-Based Signals

Multichannel sample-based and frame-based signals can be buffered into multichannel frame-based signals using the Buffer block.

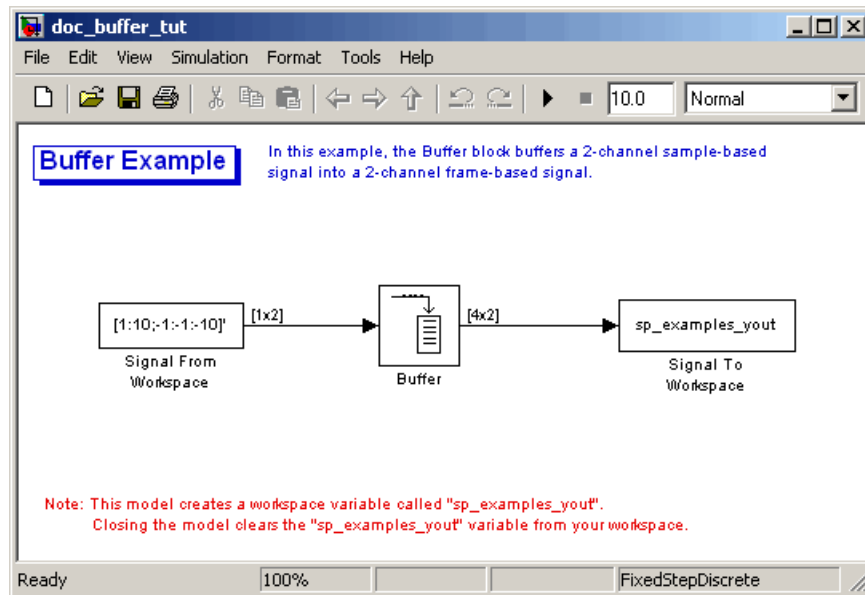
The following figure is a graphical representation of a sample-based signal being converted into a frame-based signal by the Buffer block.



In the following example, a two-channel sample-based signal is buffered into a two-channel frame-based 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 **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 sample-based 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 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 **Block Parameters: Buffer** dialog box opens.

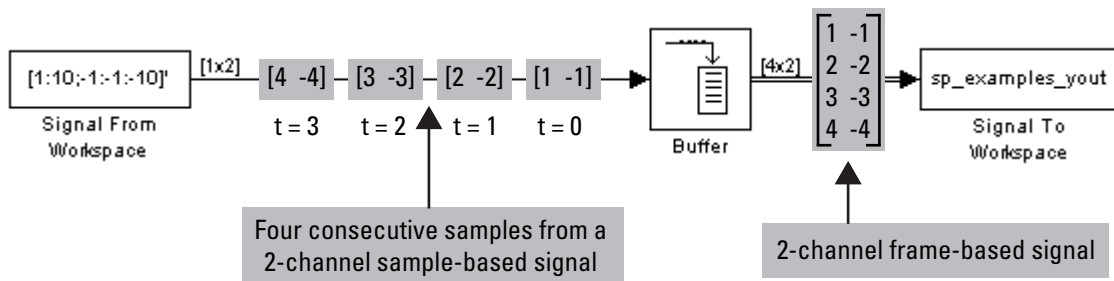
- 6 Set the parameters as follows:
- **Output buffer size (per channel)** = 4
 - **Buffer overlap** = 0
 - **Initial conditions** = 0

Because you set the **Output buffer size** parameter to 4, the Buffer block outputs a frame-based signal with frame size 4.

- 7 Run the model.

Note that the input to the Buffer block is sample based (represented as a single line) while the output is frame-based (represented by a double line).

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 frame-based 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-based signal.

Buffer Sample-Based Signals into Frame-Based Signals with Overlap

In some cases it is useful to work with data that represents overlapping sections of an original sample-based or frame-based 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.

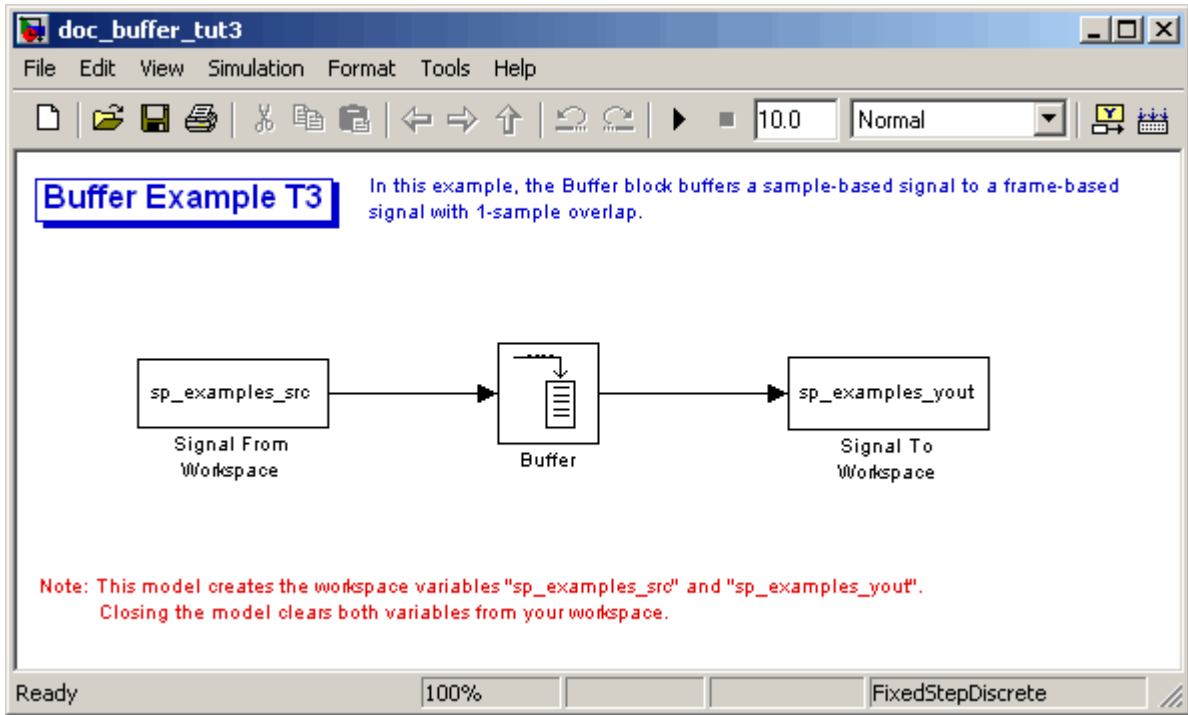
The **Buffer overlap** parameter of the Buffer block specifies the number of overlap points, L . In the overlap case ($L > 0$), the frame period for the output is $(M_o - L) * T_{si}$, where T_{si} is the input sample period and M_o is the **Buffer size**.

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 $T_{si} * (M_o - L)$, 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 sample-based signal with sample period 1 is buffered to a frame-based 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 **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 sample-based 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.

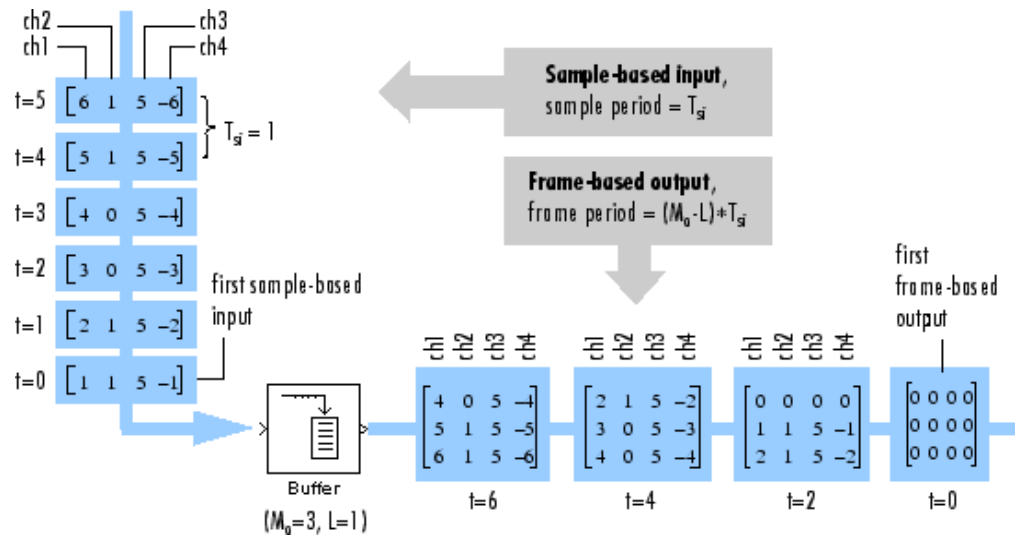
- 4 Save these parameters and close the dialog box by clicking **OK**.
- 5 Double-click the Buffer block. The **Block Parameters: Buffer** dialog box opens.
- 6 Set the block parameters as follows, and then click **OK**:
 - **Output buffer size (per channel)** = 3
 - **Buffer overlap** = 1
 - **Initial conditions** = 0

Because you set the **Output buffer size** parameter to 3, the Buffer block outputs a frame-based 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.

Note that the input to the Buffer block is sample based (represented as a single line) while the output is frame based (represented by a double line).

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.

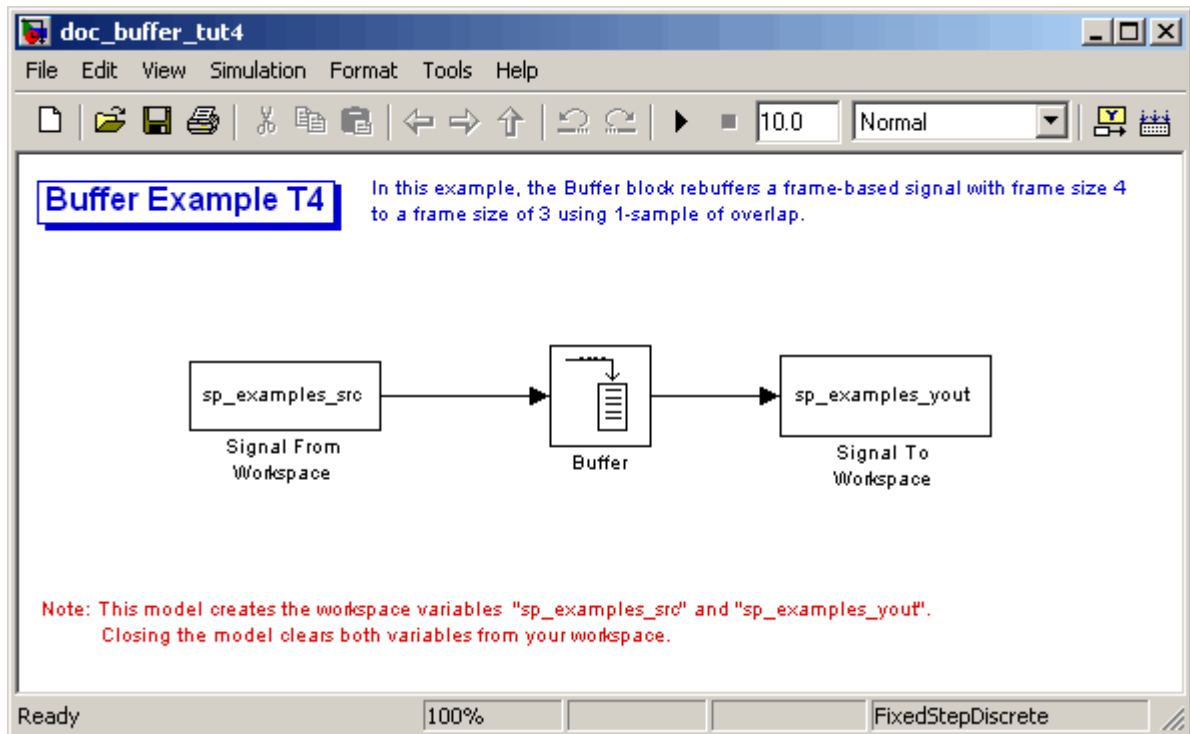
See “Excess Algorithmic Delay (Tasking Latency)” on page 2-63 for general information about algorithmic delay. For instructions on how to calculate buffering delay, see “Buffer Delay and Initial Conditions” on page 2-50.

Buffer Frame-Based Signals into Other Frame-Based Signals

In the following example, a two-channel frame-based signal with frame size 4 is rebuffered to a frame-based 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 **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

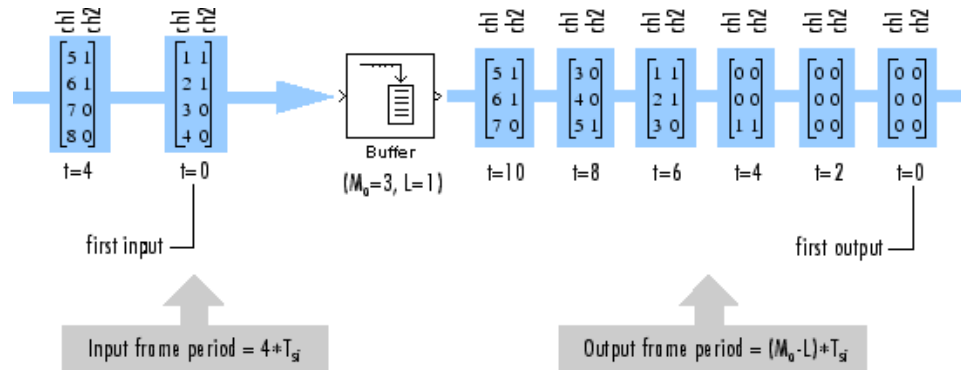
Based on these parameters, the Signal From Workspace block outputs a two-channel, frame-based signal with a sample period of 1 second and a frame size of 4.

- 4 Save these parameters and close the dialog box by clicking **OK**.
- 5 Double-click the Buffer block. The **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-based 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 2-63 for general information about algorithmic delay. For instructions on how to calculate buffering delay, and see “Buffer Delay and Initial Conditions” on page 2-50.

Buffer Delay and Initial Conditions

In the examples “Buffer Sample-Based Signals into Frame-Based Signals with Overlap” on page 2-42 and “Buffer Frame-Based Signals into Other Frame-Based Signals” on page 2-47, 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-Based Signals into Other Frame-Based Signals” on page 2-47 using the following command at the MATLAB command line:

```
d = rebuffer_delay(4,3,1)
d = 8
```

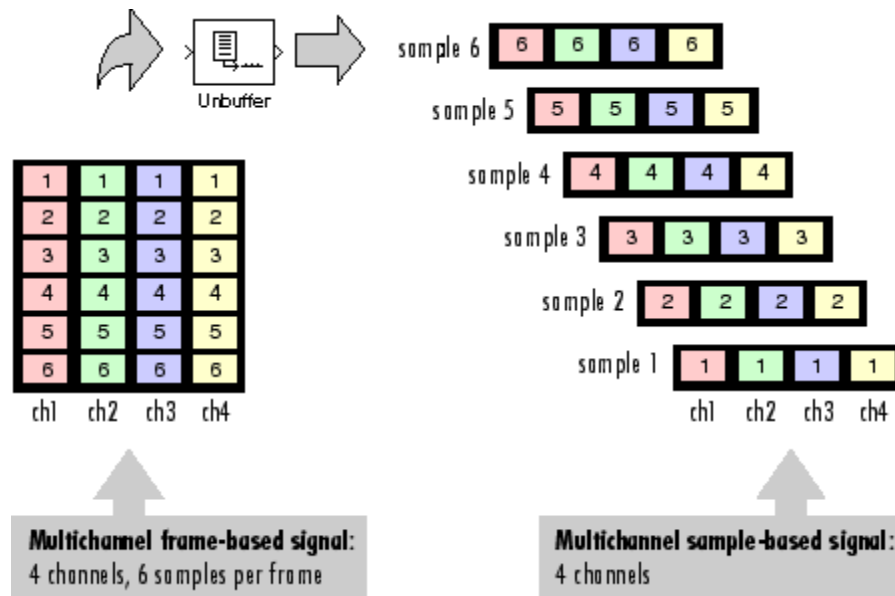
This result agrees with the block’s output in that example. Notice that this model was simulated in Simulink multitasking mode.

For more information about delay, see “Excess Algorithmic Delay (Tasking Latency)” on page 2-63. 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-Based Signals into Sample-Based Signals

You can unbuffer multichannel frame-based signals into multichannel sample-based signals using the Unbuffer block. The Unbuffer block performs the inverse operation of the Buffer block's "sample-based to frame-based" buffering process, and generates an N-channel sample-based output from an N-channel frame-based input. The first row in each input matrix is always the first sample-based output.

The following figure is a graphical representation of this process.



The sample period of the sample-based output, T_{so} , is related to the input frame period, T_{fi} , by the input frame size, M_i .

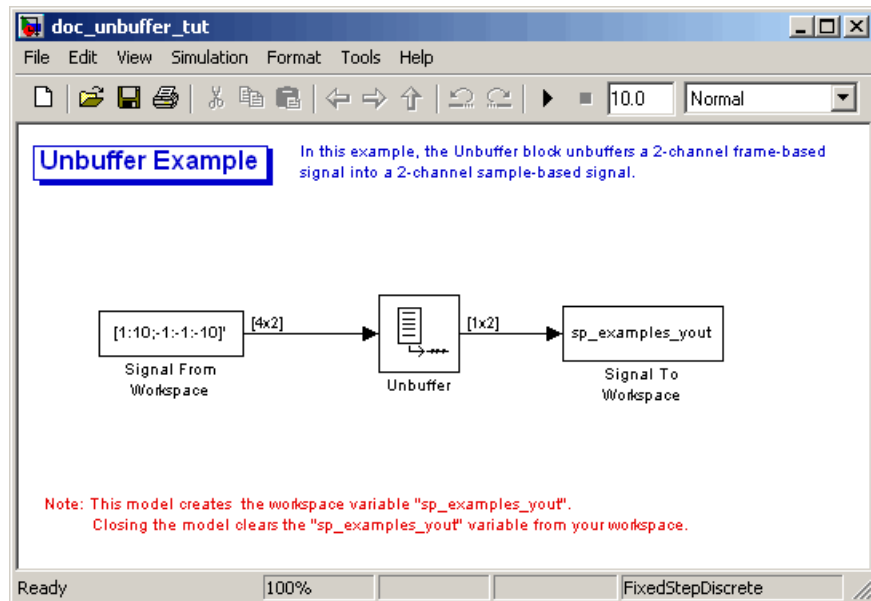
$$T_{so} = T_{fi} / M_i$$

The Unbuffer block always preserves the signal's sample period ($T_{so} = T_{si}$). See "Convert Sample and Frame Rates in Simulink" on page 2-17 for more information about rate conversions.

In the following example, a two-channel frame-based signal is unbuffered into a two-channel sample-based signal:

- 1 At the MATLAB command prompt, type `ex_unbuffer_tut`.

The Unbuffer Example model opens.



- 2 Double-click the Signal From Workspace block. The **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

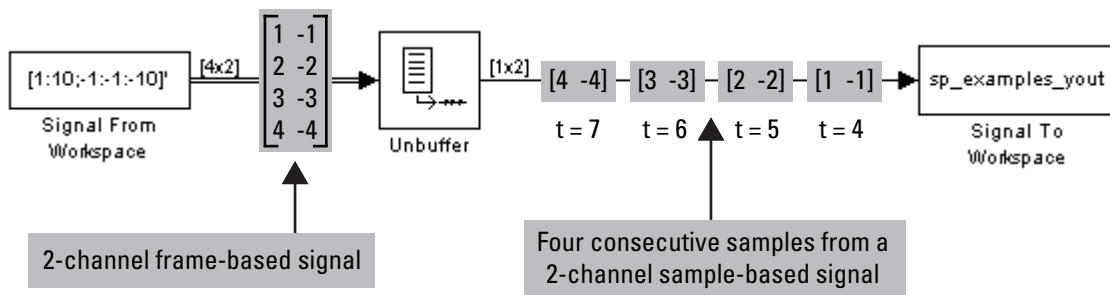
Based on these parameters, the Signal From Workspace block outputs a two-channel, frame based-signal with frame size 4.

- 4 Save these parameters and close the dialog box by clicking **OK**.
- 5 Double-click the Unbuffer block. The **Block Parameters: Unbuffer** dialog box opens.
- 6 Set the **Initial conditions** parameter to 0, and then click **OK**.

The Unbuffer block unbuffers the frame-based signal into a two-channel sample-based signal.

- 7 Run the model.

The following figure 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) =
    0    0
```

```
sp_examples_yout(:,:,2) =
```

```
    0    0
```

```
sp_examples_yout(:,:,3) =
```

```
    0    0
```

```
sp_examples_yout(:,:,4) =
```

```
    0    0
```

```
sp_examples_yout(:,:,5) =
```

```
    1   -1
```

```
sp_examples_yout(:,:,6) =
```

```
    2   -2
```

```
sp_examples_yout(:,:,7) =
```

```
    3   -3
```

The Unbuffer block unbuffers the frame-based signal into a two-channel, sample-based signal. Each page of the output matrix represents a different sample time.

Delay and Latency

Note Starting in R2010b, many DSP System Toolbox blocks received a new parameter to control whether they perform sample- or frame-based processing. The following content has not been updated to reflect this change. For more information, see the “Frame-Based Processing” section of the Release Notes.

In this section...
“Computational Delay” on page 2-55
“Algorithmic Delay” on page 2-57
“Zero Algorithmic Delay” on page 2-57
“Basic Algorithmic Delay” on page 2-60
“Excess Algorithmic Delay (Tasking Latency)” on page 2-63
“Predict Tasking Latency” on page 2-65

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 “Improving Simulation Performance and Accuracy” in the Simulink documentation, which describes some basic strategies. The following information discusses several additional 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 2-6 for more information.
- Use the `dspstartup` file to tailor Simulink for signal processing models, or manually make the adjustments described in “Settings in `dspstartup.m`” in the *DSP System Toolbox Getting Started Guide*.
- 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 M-file 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:

- “Zero Algorithmic Delay” on page 2-57
- “Basic Algorithmic Delay” on page 2-60
- “Excess Algorithmic Delay (Tasking Latency)” on page 2-63

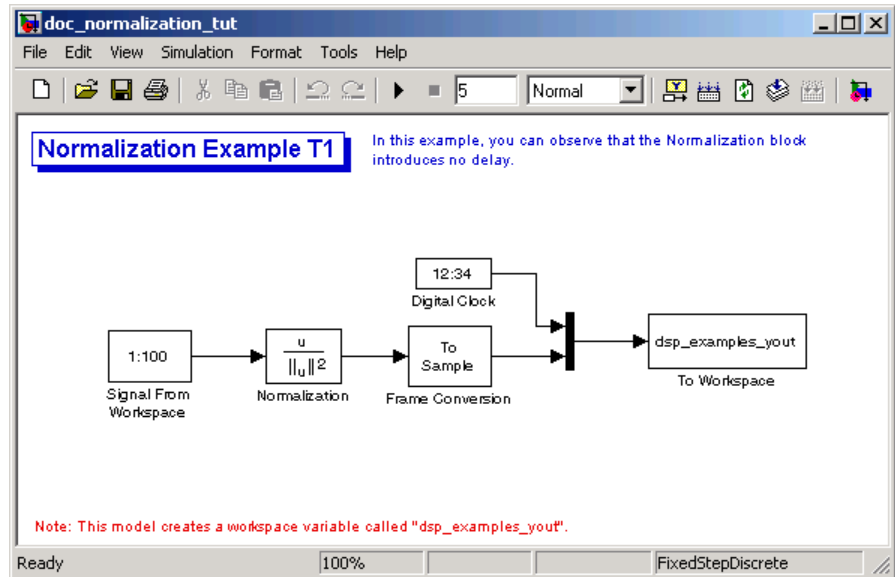
The following topics explain the different categories of delay, and how the simulation and parameter settings can affect the level of delay that a particular block experiences.

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 **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 Double-click the Frame Conversion block. The **Block Parameters: Frame Conversion** dialog box opens.
- 6 Set the **Output signal** parameter to **Sample based**, and then click **OK**.
- 7 Run the model.

The model prepends the current value of the Simulink timer output from the Digital Clock block to each output frame. The Frame Conversion block converts the frame-based signal to a sample-based signal so that the output in the MATLAB Command Window is more easily readable.

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]'
```

8 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

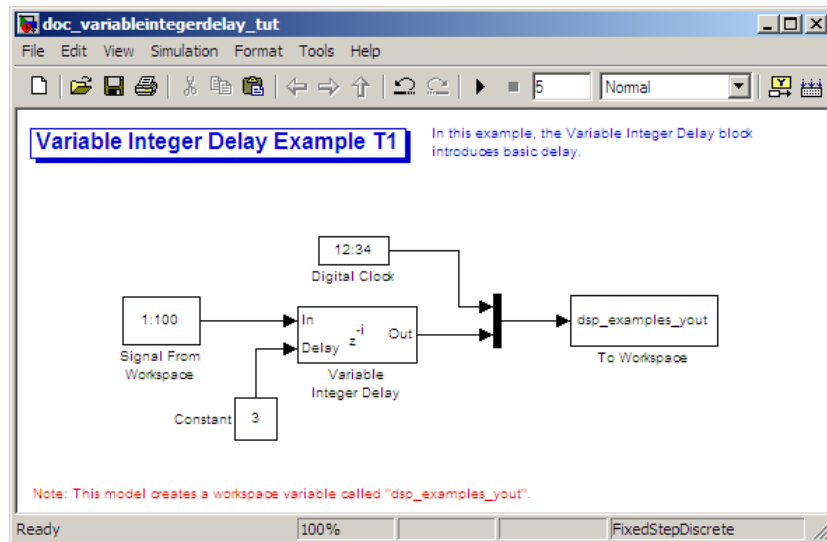
When several blocks with zero algorithmic delay are connected in a feedback loop, Simulink may report an algebraic loop error and performance may generally suffer. You can prevent algebraic loops by injecting at least one sample of delay into a feedback loop, for example, by including a Delay block with **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 **Block Parameters: Signal From Workspace** dialog box opens.
- 3 Set the block parameters as follows:

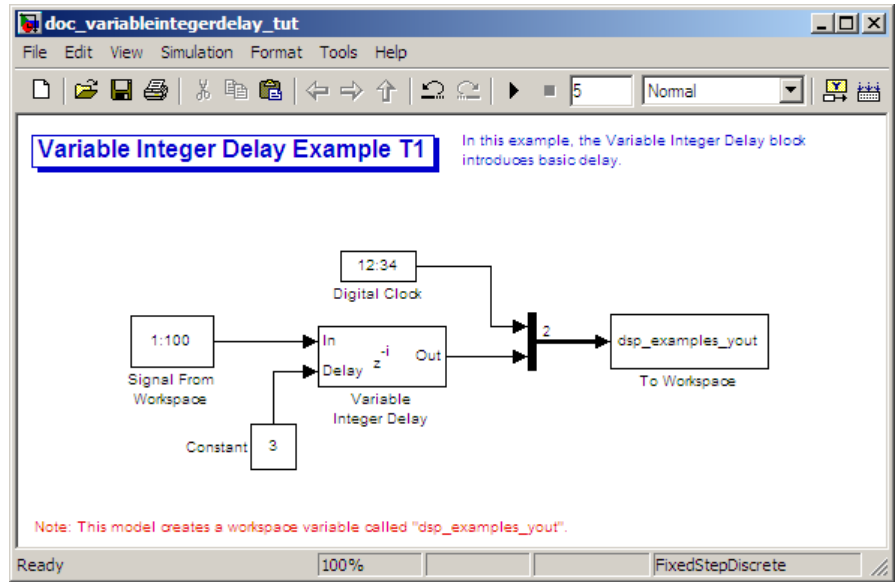
- **Signal** = 1:100
 - **Sample time** = 1
 - **Samples per frame** = 1
- 4** Save these parameters and close the dialog box by clicking **OK**.
 - 5** Double-click the Constant block. The **Block Parameters: Constant** dialog box opens.
 - 6** Set the block parameters as follows:
 - **Constant value** = 3
 - **Interpret vector parameters as 1-D** = Clear this check box
 - **Sampling mode** = Sample based
 - **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 **Block Parameters: Variable Integer Delay** dialog box opens.
- 8** Set the **Initial conditions** parameter to -1, and then click **OK**.
- 9** From the **Format** menu, point to **Port/Signal Displays**, 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.

$$\text{Algorithmic delay} = \text{Basic algorithmic delay} + \text{Tasking latency}$$

The tasking latency for a particular block may be dependent on the following block and model characteristics:

- “Simulink Tasking Mode” on page 2-63
- “Block Rate Type” on page 2-64
- “Model Rate Type” on page 2-64
- “Block Sample Mode” on page 2-65

Simulink Tasking Mode

Simulink has two tasking modes:

- Single-tasking
- Multitasking

To select a mode, from the **Simulation** menu, select **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 “Scheduling” in the *Simulink Coder User’s Guide*.

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 Sample Mode

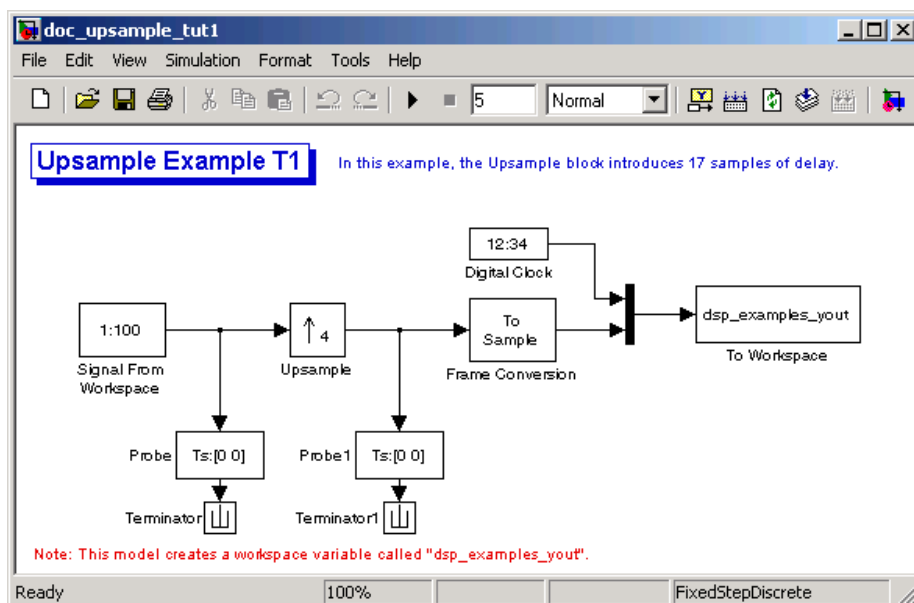
Many blocks can operate in either sample-based or frame-based modes. In source blocks, the mode is usually determined by the **Samples per frame** parameter. If, for the **Samples per frame** parameter, you enter 1, the block operates in sample-based mode. If you enter a value greater than 1, the block operates in frame-based mode. In nonsource blocks, the sample mode is determined by the input signal. See the block reference pages for additional information about specific blocks.

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 **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 **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

- 7 Double-click the **Upsample** block. The **Block Parameters: Upsample** dialog box opens.

- 8 Set the block parameters as follows, and then click **OK**:
 - **Upsample factor** = 4
 - **Sample offset** = 0
 - **Initial condition** = -1
 - **Rate options** = Allow multirate processing

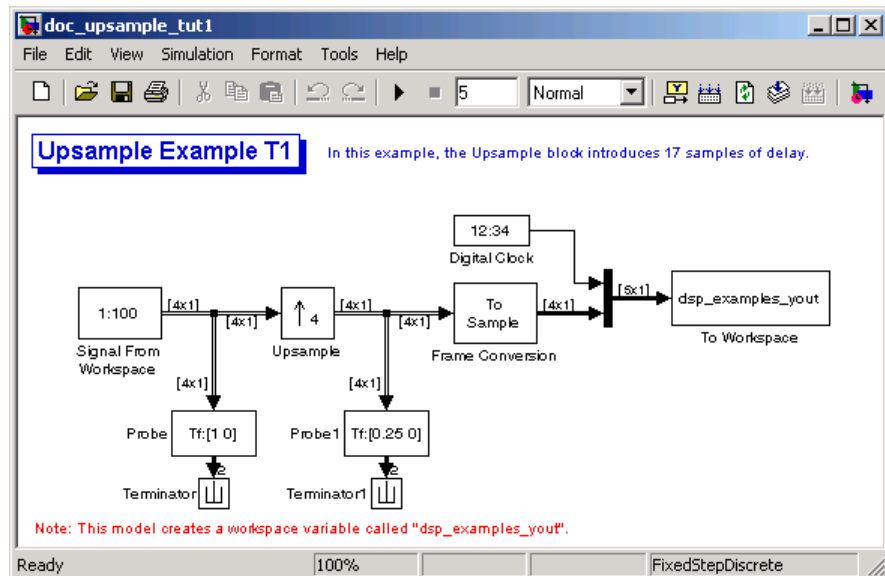
The **Rate options** parameter makes the model multirate, since the input and output frame rates will not be equal.

- 9 Double-click the **Digital Clock** block. The **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 Double-click the Frame Conversion block. The **Block Parameters: Frame Conversion** dialog box opens.
- 12 Set the **Output signal** parameter to **Sample based**, and then click **OK**.
- 13 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. The Frame Conversion block converts the frame-based signal into a sample-based signal so that the output in the MATLAB Command Window is easily readable.

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 $T_{fo} = 0.25$.

14 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 frame-based 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.

- 15** From the **Simulation** menu, select **Configuration Parameters**.
- 16** In the **Solver** pane, from the **Type** list, select **Fixed-step**. From the **Solver** list, select **Discrete** (no continuous states).
- 17** From the **Tasking mode for periodic sample times** list, select **SingleTasking**.
- 18** Run the model.

The model now runs in single-tasking mode.

- 19** 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.

Filter Analysis, Design, and Implementation

- “Design a Filter in Fdesign — Process Overview” on page 3-2
- “Design a Filter in the Filterbuilder GUI” on page 3-11
- “Use FDATool with DSP System Toolbox Software” on page 3-16
- “Digital Frequency Transformations” on page 3-87
- “Digital Filter Design Block” on page 3-122
- “Filter Realization Wizard” on page 3-134
- “Digital Filter Block” on page 3-147
- “Analog Filter Design Block” on page 3-159
- “Fixed-Point Filter Design” on page 8-65

Design a Filter in Fdesign – Process Overview

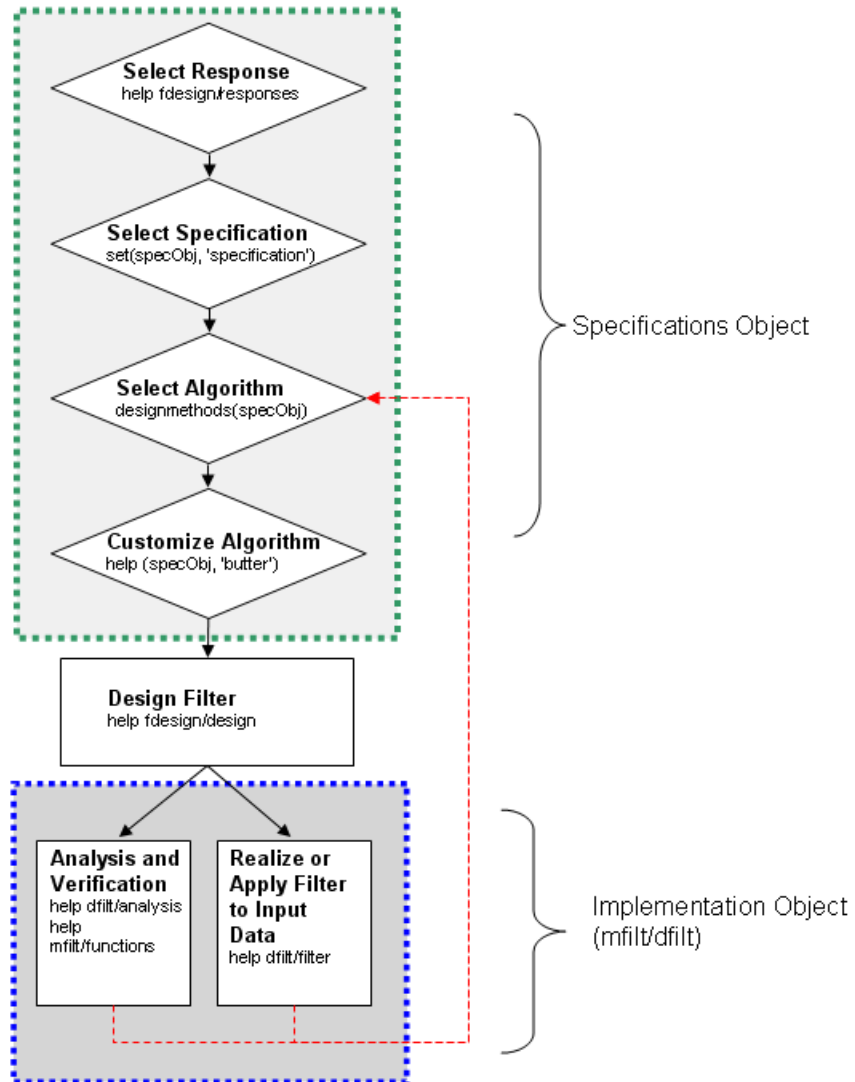
Process Flow Diagram and Filter Design Methodology

- “Exploring the Process Flow Diagram” on page 3-2
- “Select a Response” on page 3-4
- “Select a Specification” on page 3-4
- “Select an Algorithm” on page 3-6
- “Customize the Algorithm” on page 3-8
- “Design the Filter” on page 3-8
- “Design Analysis” on page 3-9
- “Realize or Apply the Filter to Input Data” on page 3-10

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 3-4, 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 3-6, but also on the specification selected in “Select a Specification” on page 3-4. To explore all the available options, type the following at the MATLAB command prompt:

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

where `d` is the 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 3-8, 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 3-6, 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 display help for analysis of a multirate filter, type:

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

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

Realize or Apply the Filter to Input Data

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

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

This step is never automatically performed for you. 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” on page 3-11
- “Filterbuilder Design Process” on page 3-11
- “Select a Response” on page 3-12
- “Select a Specification” on page 3-13
- “Select an Algorithm” on page 3-13
- “Customize the Algorithm” on page 3-13
- “Analyze the Design” on page 3-13
- “Realize or Apply the Filter to Input Data” on page 3-14

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 “Designing a Filter in Fdesign — Process Overview” in the Getting Started guide. The idea is to choose the constraints and specifications of the filter, and to use those as a starting point in the design. Postponing the choice of algorithm for the filter allows the best design method to be determined automatically, based upon the desired performance

criteria. The following are the details of each of the steps for designing a filter with `filterbuilder`.

Select a Response

When you open the `filterbuilder` tool by typing:

```
filterbuilder
```

at the MATLAB command prompt, the **Response Selection** dialog box appears, listing all possible filter responses available in 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

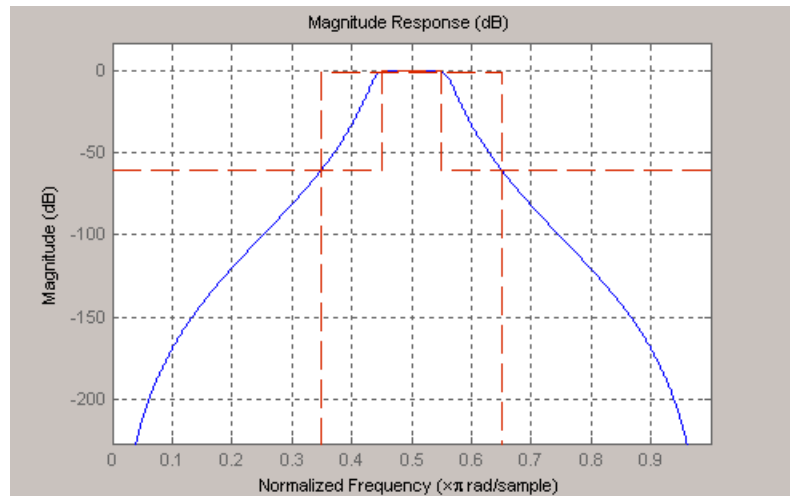
The algorithms available for your filter depend upon the filter response and design parameters you have selected in the previous steps. For example, in the case of a bandpass filter, if the impulse response selected is IIR and the **Order Mode** field is set to **Minimum**, the design methods available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

Customize the Algorithm

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

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

In this section...

- “Design Advanced Filters in FDATool” on page 3-16
- “Access the Quantization Features of FDATool” on page 3-20
- “Quantize Filters in FDATool” on page 3-22
- “Analyze Filters in MATLAB with a Noise-Based Method” on page 3-30
- “Scale Second-Order Section Filters” on page 3-37
- “Reorder the Sections of Second-Order Section Filters” on page 3-42
- “View SOS Filter Sections” on page 3-48
- “Import and Export Quantized Filters” on page 3-53
- “Generate MATLAB Code” on page 3-58
- “Import XILINX Coefficient (.COE) Files” on page 3-59
- “Transform Filters Using FDATool” on page 3-59
- “Design Multirate Filters in FDATool” on page 3-69
- “Realize Filters as Simulink Subsystem Blocks” on page 3-83

Design Advanced Filters in FDATool

- “Overview of FDATool Features” on page 3-16
- “Use FDATool with DSP System Toolbox Software” on page 3-17
- “Design a Notch Filter” on page 3-18

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 **10000**.
- 4** Set the location of the center of the notch, in either normalized frequency or **Hz**. For the notch center at 440 Hz, enter **440**.

5 To shape the notch, enter the **bandwidth**, `bw`, to be 40.

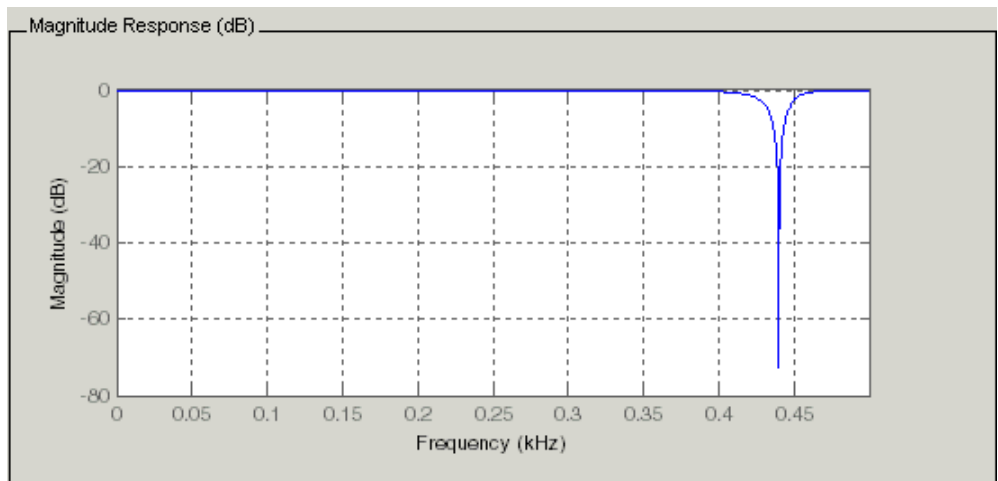
6 Leave the **Magnitude Specification** in dB (the default) and leave **Apass** as 1.

7 Click Design Filter.

FDATool computes the filter coefficients and plots the filter magnitude response in the analysis area for you to review.

When you design a single notch filter, you do not have the option of setting the filter order — the **Filter Order** options are disabled.

Your filter should look about like this:



For more information about a design method, refer to the online Help system. For instance, to get further information about the **Q** setting for the notch filter in FDATool, enter

```
doc iirnotch
```

at the 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.

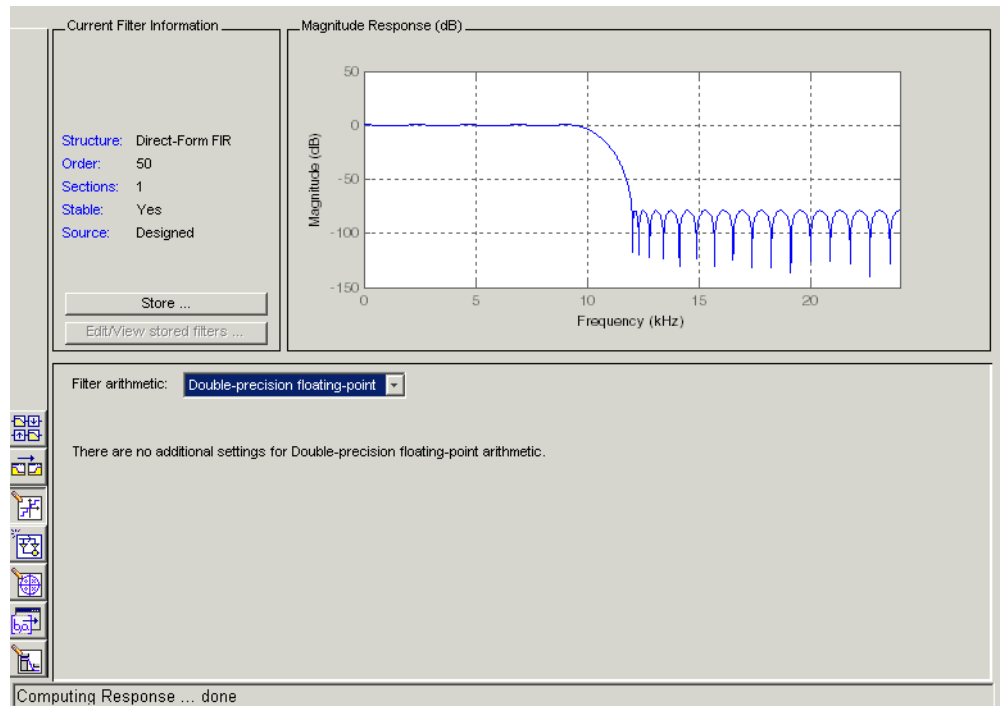
Any one of the designs may be quantized in FDATool and analyzed with the available analyses on the **Analysis** menu.

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” on page 3-22
- “Coefficients Options” on page 3-23
- “Input/Output Options” on page 3-25
- “Filter Internals Options” on page 3-26
- “Filter Internals Options for CIC Filters” on page 3-29

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.

Option Name	When Used	Description
Denominator Range (+/-)	IIR filters	Lets you set the range the denominator coefficients represent. You use this instead of the Denominator Frac. Length option to set the precision. When you enter a value x, the resulting range is -x to x. Range must be a positive integer.
Best-precision fraction lengths	All filters	Directs FDATool to select the fraction lengths for numerator (and denominator where available) values to maximize the filter performance. Selecting this option disables all of the fraction length options for the filter.
Scale Values frac. length	SOS IIR filters	Sets the fraction length used to interpret the scale values in SOS filters.
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 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.

Option Name	When Used	Description
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 .
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	<p>Sets the mode the filter uses to quantize numeric values when the values lie between representable values for the data format (word and fraction lengths). Choose from one of:</p> <ul style="list-style-type: none"> • ceil - Round toward positive infinity. • convergent - Round to the closest representable integer. Ties round to the nearest even stored integer. This

Option	Equivalent Filter Property (Using Wildcard *)	Description
		<p>is the least biased of the methods available in this software.</p> <ul style="list-style-type: none"> • <code>fix/zero</code> - Round toward zero. • <code>floor</code> - Round toward negative infinity. • <code>nearest</code> - Round toward nearest. Ties round toward positive infinity. • <code>round</code> - 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 <code>saturate</code> (limit the output to the largest positive or negative representable value) or <code>wrap</code> (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 (<code>FullPrecision</code>), or whether to keep the most significant bit (<code>KeepMSB</code>) or least significant bit (<code>KeepLSB</code>) in the result when you need to shorten the 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.

Option	Equivalent Filter Property (Using Wildcard *)	Description
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.
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		

Option	Equivalent Filter Property (Using Wildcard *)	Description
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 in MATLAB with a Noise-Based Method

- “Analyze Filters with the Magnitude Response Estimate Method” on page 3-31

- “Compare the Estimated and Theoretical Magnitude Responses” on page 3-35
- “Select Quantized Filter Structures” on page 3-35
- “Convert the Structure of a Quantized Filter” on page 3-35
- “Convert Filters to Second-Order Sections Form” on page 3-36

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 F_s . The first time you run the analysis, magnitude response estimate uses default settings for the various conditions that define the process, such as the number of test points and the number of trials.

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 $F_s/2$	Frequency range of the plot x-axis.
Frequency Units	Hz	Units for specifying the frequency range.
Sampling Frequency	48000	Inverse of the sampling period.

Analysis Parameter	Default Setting	Description
Frequency Scale	dB	Units used for the y-axis display of the output.
Normalized Frequency	Off	Use normalized frequency for the display.

After your first analysis run ends, open the **Analysis Parameters** dialog 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.

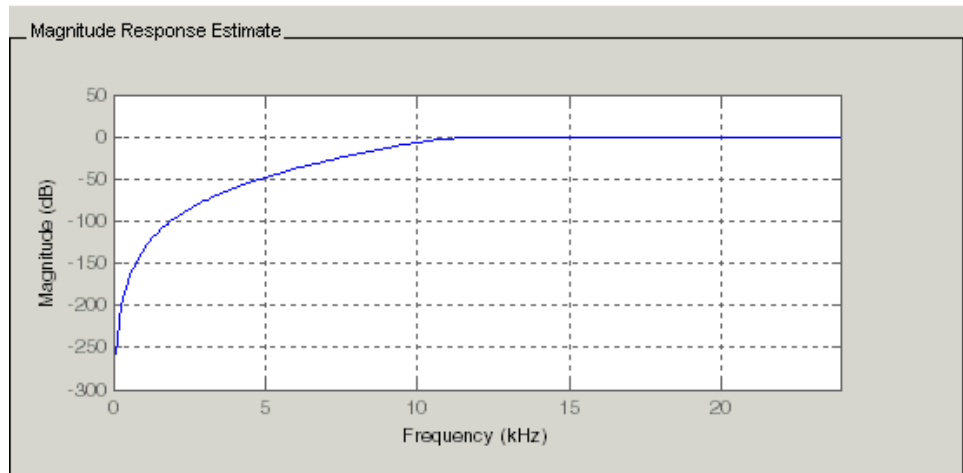
FDATool switches to quantization mode and displays the quantization panel.

- 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 3-32, 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 3-31.

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” 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” 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 3-37
- “Scale an SOS Filter” on page 3-39

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 <i>norm</i> for details.
Maximum Numerator	Maximum allowed value for numerator coefficients after scaling.
Numerator Constraint	Specifies whether and how to constrain numerator coefficient values. Options are none, normalize, power of 2, and unit. Choosing none lets the scaling use any scale value for the numerators by removing any constraints on the numerators, except that the coefficients will be clipped if they exceed the Maximum Numerator . With Normalize the maximum absolute value of the numerator is forced to equal the Maximum Numerator value (for all other constraints, the Maximum Numerator is only an upper limit, above which coefficients will be clipped). The power of 2 option forces scaling to use numerator values that are powers of 2, such as 2 or 0.5. With unit, the leading coefficient of each numerator is forced to a value of 1.

Parameter	Description and Valid Value
Overflow Mode	Sets the way the filter handles arithmetic overflow situations during scaling. Choose from either saturate (limit the output to the largest positive or negative representable value) or wrap (set overflowing values to the nearest representable value using modular arithmetic).
Scale Value Constraint	Specify whether to constrain the filter scale values, and how to constrain them. Valid options are unit , power of 2 , and none . Choosing unit for the constraint disables the Max. Scale Value setting and forces scale values to equal 1. Power of 2 constrains the scale values to be powers of 2, such as 2 or 0.5, while none removes any constraint on the scale values, except that they cannot exceed the Max. Scale Value .
Max. Scale Value	Sets the maximum allowed scale values. SOS filter scaling applies the Max. Scale Value limit only when you set Scale Value Constraint to a value other than unit (the default setting). Setting a maximum scale value removes any other limits on the scale values.
Revert to Original Filter	Returns your filter to the original scaling. Being able to revert to your original filter makes it easier to assess the results of scaling your filter.

Various combinations of settings let you scale filter numerators without changing the scale values, or adjust the filter scale values without changing the numerators. There is no scaling control for denominators.

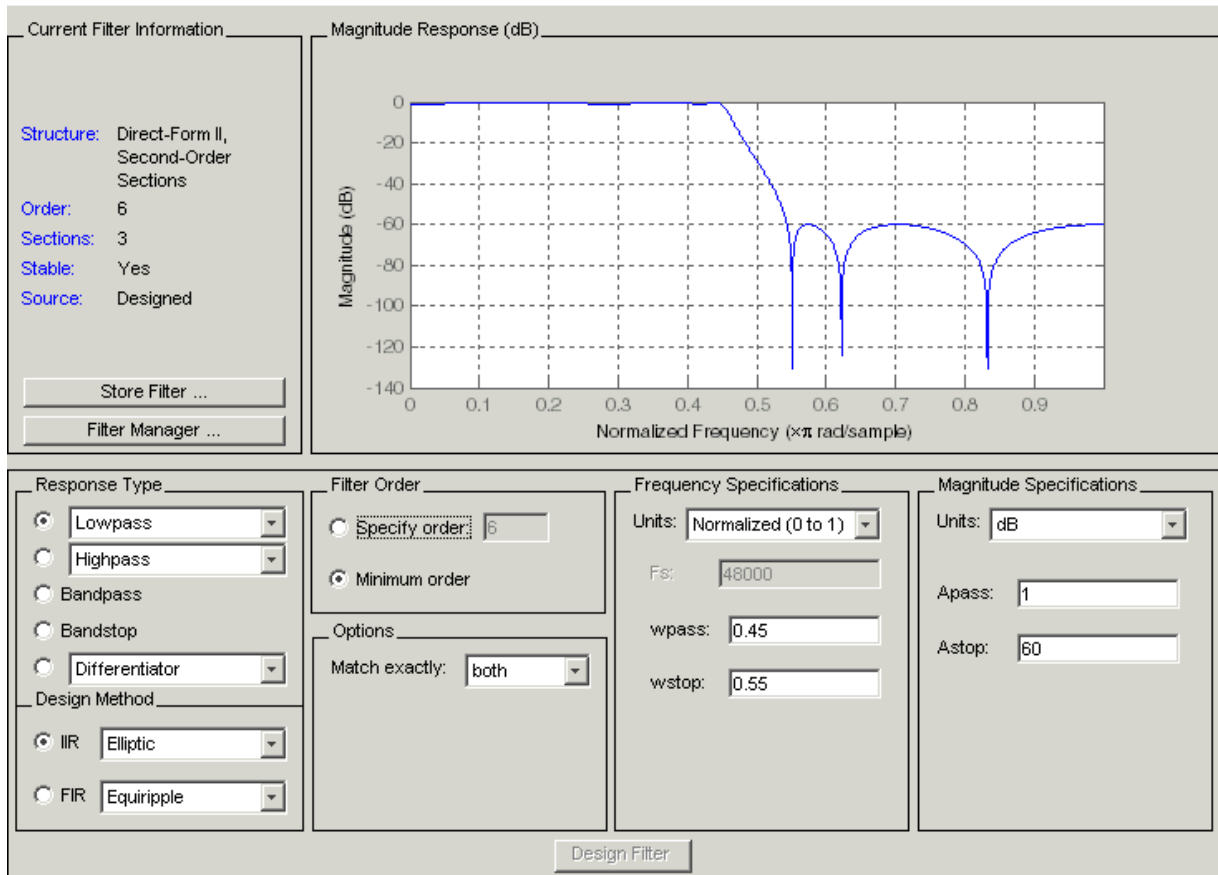
Scale an SOS Filter

Start the process by designing a lowpass elliptical filter in FDATool.

- 1 Launch FDATool.
- 2 In **Response Type**, select **Lowpass**.

- 3 In Design Method, select **IIR** and **Elliptic** from the IIR design methods list.
- 4 Select **Minimum Order** for the filter.
- 5 Switch the frequency units by choosing **Normalized(0 to 1)** from the **Units** list.
- 6 To set the passband specifications, enter **0.45** for **wpass** and **0.55** for **wstop**. Finally, in **Magnitude Specifications**, set **Astop** to **60**.
- 7 Click **Design Filter** to design the filter.

After FDATool finishes designing the filter, you see the following plot and settings in the tool.



You kept the **Options** setting for **Match exactly** as both, meaning the filter design matches the specification for the passband and the stopband.

- 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.

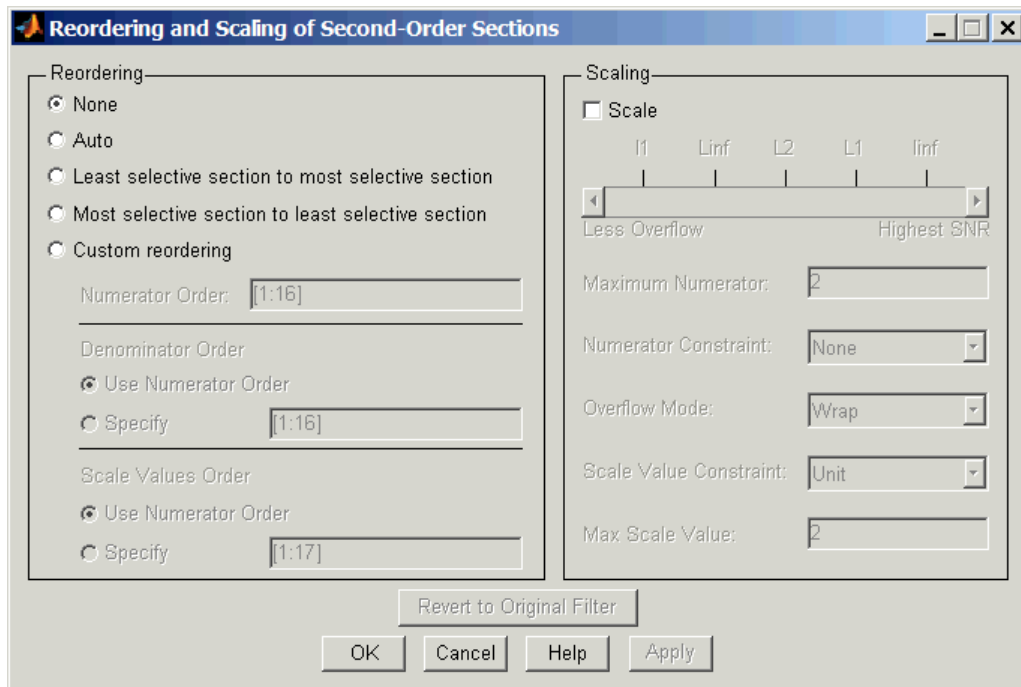
Reorder the Sections of Second-Order Section Filters

Reorder Filters Using FDATool

FDATool design most discrete-time filters in second-order sections. Generally, SOS filters resist the effects of quantization changes when you create fixed-point filters. After you have a second-order section filter in FDATool, either one you designed in the tool, or one you imported, FDATool provides the capability to change the order of the sections that compose the filter. Any SOS filter in FDATool allows reordering of the sections.

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

With your SOS filter in FDATool, select **Edit > Reorder and Scale** from the menu bar. FDATool returns the reordering dialog box shown here with the default settings.



Controls on the Reordering and Scaling of Second-Order Sections dialog box

In this dialog box, the left-hand side contains options for reordering SOS filters. On the right you see the scaling options. These are independent — reordering your filter does not require scaling (note the **Scale** option) and scaling does not require that you reorder your filter (note the **None** option under **Reordering**). For more about scaling SOS filters, refer to “Scale Second-Order Section Filters” on page 3-37 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. Automatic ordering adapts to the specification type of your filter.
None	Does no reordering on your filter. Selecting None lets you scale your filter without applying reordering at the same time. When you access this dialog box with a current filter, this is the default setting — no reordering is applied.
Least selective section to most selective section	Rearranges the filter sections so the least restrictive (lowest Q) section is the first section and the most restrictive (highest Q) section is the last section.
Most selective section to least selective section	Rearranges the filter sections so the most restrictive (highest Q) section is the first section and the least restrictive (lowest Q) section is the last section.
Custom reordering	Lets you specify the section ordering to use by enabling the Numerator Order and Denominator Order options
Numerator Order	Specify new ordering for the sections of your SOS filter. Enter a vector of the indices of the sections in the order in which to rearrange them. For example, a filter with five sections has indices 1, 2, 3, 4, and 5. To switch the second and fourth sections, the vector would be [1,4,3,2,5].
Use Numerator Order	Rearranges the denominators in the order assigned to the numerators.

Control Option	Description
Specify	Lets you specify the order of the denominators, rather than using the numerator order. Enter a vector of the indices of the sections to specify the order of the denominators to use. For example, a filter with five sections has indices 1, 2, 3, 4, and 5. To switch the second and fourth sections, the vector would be [1,4,3,2,5].
Use Numerator Order	Reorders the scale values according to the order of the numerators.
Specify	Lets you specify the order of the scale values, rather than using the numerator order. Enter a vector of the indices of the sections to specify the order of the denominators to use. For example, a filter with five sections has indices 1, 2, 3, 4, and 5. To switch the second and fourth sections, the vector would be [1,4,3,2,5].
Revert to Original Filter	Returns your filter to the original section ordering. Being able to revert to your original filter makes comparing the results of changing the order of the sections easier to assess.

Reorder an SOS Filter. With FDATool open and a second-order filter as the current filter, you use the following process to access the reordering capability and reorder you filter. Start by launching FDATool from the command prompt.

- 1 Enter `fdatool` at the command prompt to launch FDATool.
- 2 Design a lowpass Butterworth filter with order 10 and the default frequency specifications by entering the following settings:
 - Under **Response Type** select Lowpass.
 - Under **Design Method**, select **IIR** and Butterworth from the list.
 - Specify the order equal to 10 in **Specify order** under **Filter Order**.
 - Keep the default **Fs** and **Fc** values in **Frequency Specifications**.

3 Click **Design Filter**.

FDATool design the Butterworth filter and returns your filter as a Direct-Form II filter implemented with second-order sections. You see the specifications in the **Current Filter Information** area.

With the second-order filter in FDATool, reordering the filter uses the **Reordering and Scaling of Second-Order Sections** feature in FDATool (also available in Filter Visualization Tool, fvtool).

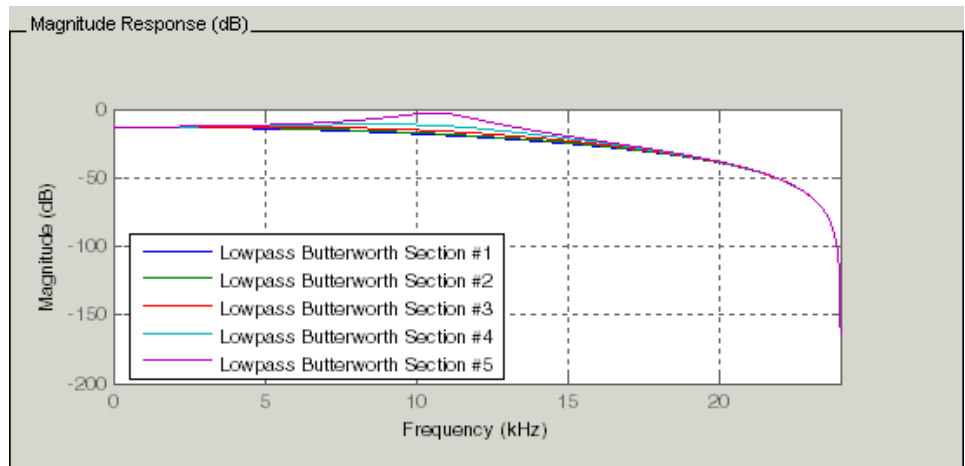
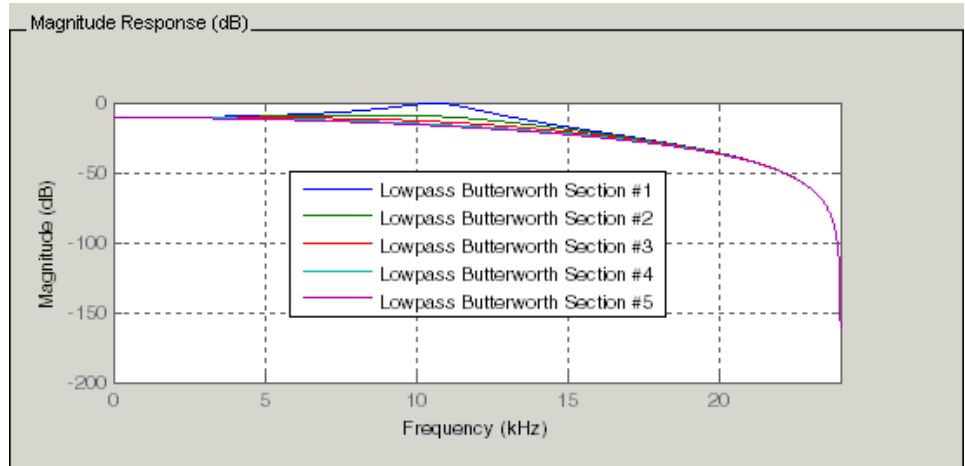
4 To reorder your filter, select **Edit > Reorder and Scale Second-Order Sections** from the FDATool menus.

Now you are ready to reorder the sections of your filter. Note that FDATool performs the reordering on the current filter in the session.

Use Least Selective to Most Selective Section Reordering. To let FDATool reorder your filter so the least selective section is first and the most selective section is last, perform the following steps in the **Reordering and Scaling of Second-Order Sections** dialog box.

- 1 In **Reordering**, select **Least selective section to most selective section**.
- 2 To prevent filter scaling at the same time, clear **Scale** in **Scaling**.
- 3 In FDATool, select **View > SOS View** from the menu bar so you see the sections of your filter displayed in FDATool.
- 4 In the **SOS View** dialog box, select **Individual sections**. Making this choice configures FDATool to show the magnitude response curves for each section of your filter in the analysis area.
- 5 Back in the **Reordering and Scaling of Second-Order Sections** dialog box, click **Apply** to reorder your filter according to the Qs of the filter sections, and keep the dialog box open. In response, FDATool presents the responses for each filter section (there should be five sections) in the analysis area.

In the next two figures you can compare the ordering of the sections of your filter. In the first figure, your original filter sections appear. In the second figure, the sections have been rearranged from least selective to most selective.



You see what reordering does, although the result is a bit subtle. Now try custom reordering the sections of your filter or using the most selective to least selective reordering option.

View SOS Filter Sections

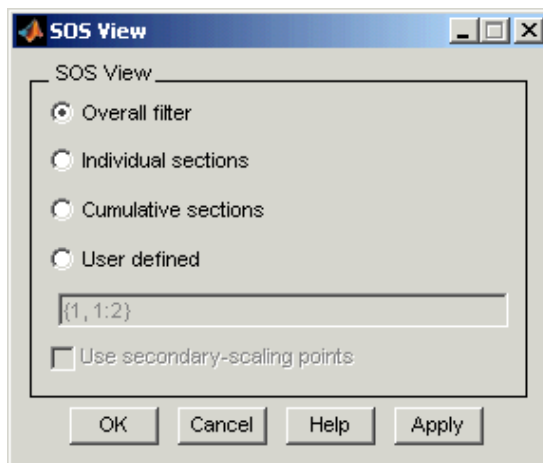
- “Using the SOS View Dialog Box” on page 3-48
- “View the Sections of SOS Filters” on page 3-50

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	<p>When you select this option, FDATool displays the response for each section as the accumulated response of all prior sections in the filter. If your filter has five sections you see five response curves:</p> <ul style="list-style-type: none"> • 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. <p>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.</p>

Option	Description
User defined	Here you define which sections to display, and in which order. Selecting this option enables the text box where you enter a cell array of the indices of the filter sections. Each index represents one section. Entering one index plots one response. Entering something like {1:2} plots the combined response of sections 1 and 2. If you have a filter with four sections, the entry {1:4} plots the combined response for all four sections, whereas {1,2,3,4} plots the response for each section. Note that after you enter the cell array, you need to click OK or Apply to update the FDATool analysis area to the new SOS View configuration.
Use secondary-scaling points	This directs FDATool to use the secondary scaling points in the sections to determine where to split the sections. This option applies only when the filter is a <code>df2sos</code> or <code>df1tsos</code> filter. For these structures, the secondary scaling points refer to the scaling locations between the recursive and the nonrecursive parts of the section (the "middle" of the section). By default, secondary-scaling points is not enabled. 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.

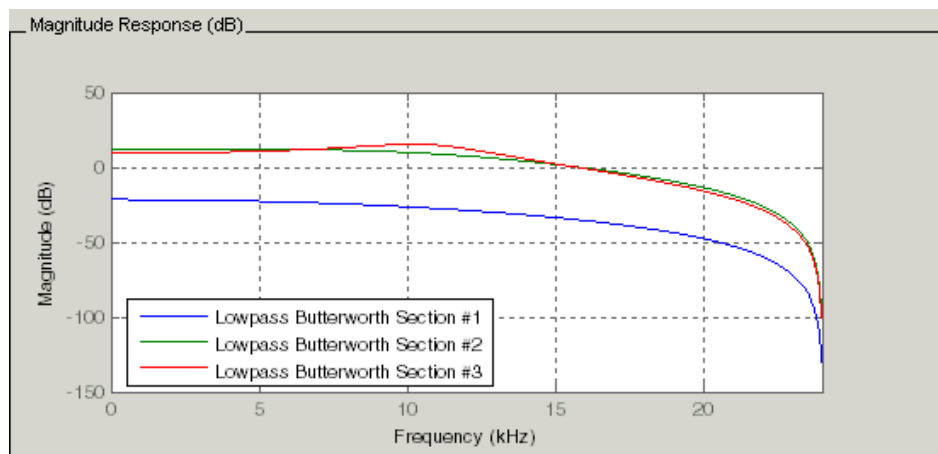
- 2 Create a lowpass SOS filter using the Butterworth design method. Specify the filter order to be 6. Using a low order filter makes seeing the sections more clear.
- 3 Design your new filter by clicking **Design Filter**.

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.

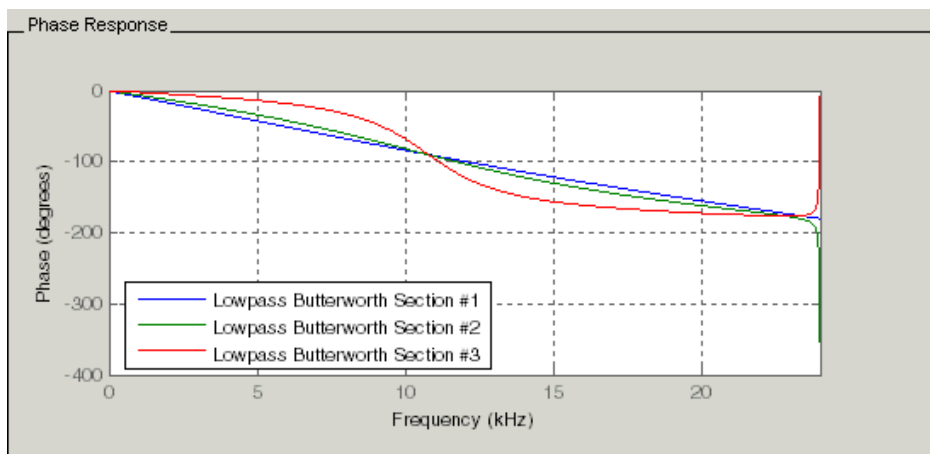
- 4 To enable SOS View, select **View > SOS View** from the menu bar.

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, you see the phase response for each filter section in the analysis area.

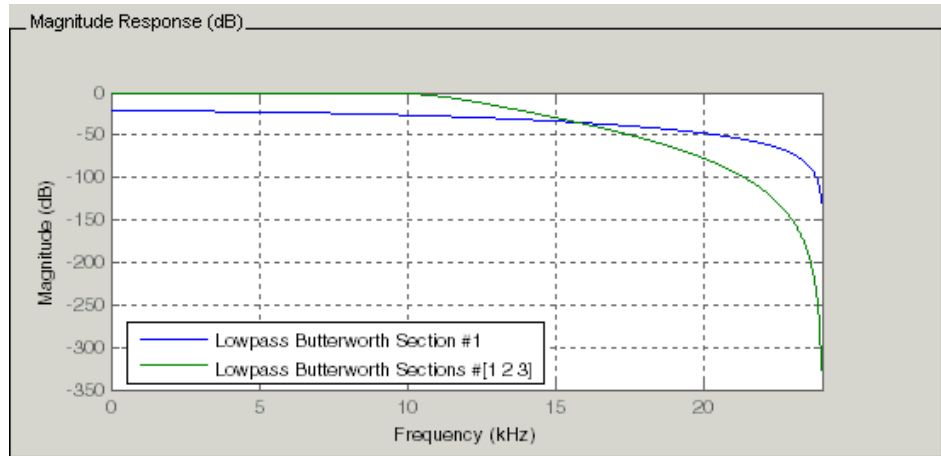


7 To define your own display of the sections, you use the **User defined** option and enter a vector of section indices to display. Now you see a display of the first section response, and the cumulative first, second, and third sections response:

- 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 3-53
- “Import Quantized Filters” on page 3-54
- “To Export Quantized Filters” on page 3-56

Overview and Structures

When you import a quantized filter into FDATool, or export a quantized filter from FDATool to your workspace, the import and export functions use objects and you specify the filter as a variable. This contrasts with importing and exporting nonquantized filters, where you select the filter structure and enter the filter numerator and denominator for the filter transfer function.

You have the option of exporting quantized filters to your MATLAB workspace, exporting them to text files, or exporting them to MAT-files.

For general information about importing and exporting filters in FDATool, refer to “FDATool: A Filter Design and Analysis GUI” in the *Signal Processing Toolbox User’s Guide*.

FDATool imports quantized filters having the following structures:

- Direct form I

- Direct form II
- Direct form I transposed
- Direct form II transposed
- Direct form symmetric FIR
- Direct form antisymmetric FIR
- Lattice allpass
- Lattice AR
- Lattice MA minimum phase
- Lattice MA maximum phase
- Lattice ARMA
- Lattice coupled-allpass
- Lattice coupled-allpass power complementary

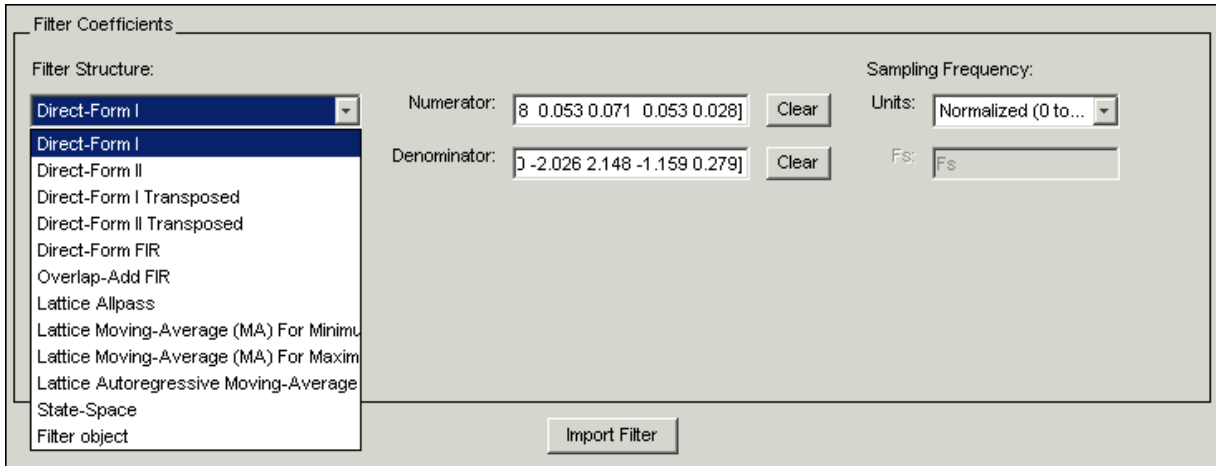
Import Quantized Filters

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 **F_s** if needed. Your sampling frequency must correspond to the units you select. For example, when you select **Normalized (0 to 1)**, **F_s** 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 3-60
- “Original Filter Type” on page 3-61

- “Frequency Point to Transform” on page 3-65
- “Transformed Filter Type” on page 3-65
- “Specify Desired Frequency Location” on page 3-66

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,  on the side bar, the **Transform Filter** panel opens in FDATool, as shown here.

Frequency Transformations

Original filter type: Lowpass

Transformed filter type: Lowpass

Frequency point to transform: 9.6 kHz

Specify desired frequency location: 7.2 kHz

Transform Filter

Your options for **Original filter type** refer to the type of your current filter to transform. If you select lowpass, you can transform your lowpass filter to another lowpass filter or to a highpass filter, or to numerous other filter formats, real and complex.

Note When your original filter is an FIR filter, both the FIR and IIR transformed filter type options appear on the **Transformed filter type** list. Both options remain active because you can apply the IIR transforms to an FIR filter. If your source is as IIR filter, only the IIR transformed filter options show on the list.

Original Filter Type

Select the magnitude response of the filter you are transforming from the list. Your selection changes the types of filters you can transform to. For example:

- When you select **Lowpass** with an IIR filter, your transformed filter type can be
 - **Lowpass**
 - **Highpass**
 - **Bandpass**
 - **Bandstop**
 - **Multiband**

- **Bandpass (complex)**
- **Bandstop (complex)**
- **Multiband (complex)**
- When you select **Lowpass** with an FIR filter, your transformed filter type can be
 - **Lowpass**
 - **Lowpass (FIR)**
 - **Highpass**
 - **Highpass (FIR) narrowband**
 - **Highpass (FIR) wideband**
 - **Bandpass**
 - **Bandstop**
 - **Multiband**
 - **Bandpass (complex)**
 - **Bandstop (complex)**
 - **Multiband (complex)**

In the following table you see each available original filter type and all the types of filter to which you can transform your original.

Original Filter	Available Transformed Filter Types
Lowpass FIR	<ul style="list-style-type: none"> • Lowpass • Lowpass (FIR) • Highpass • Highpass (FIR) narrowband • Highpass (FIR) wideband • Bandpass • Bandstop • Multiband

Original Filter	Available Transformed Filter Types
	<ul style="list-style-type: none"> • Bandpass (complex) • Bandstop (complex) • Multiband (complex)
Lowpass IIR	<ul style="list-style-type: none"> • Lowpass • Highpass • Bandpass • Bandstop • Multiband • Bandpass (complex) • Bandstop (complex) • Multiband (complex)
Highpass FIR	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • Lowpass • Highpass • Bandpass • Bandstop • Multiband • Bandpass (complex) • Bandstop (complex) • Multiband (complex)
Bandpass FIR	<ul style="list-style-type: none"> • Bandpass • Bandpass (FIR)
Bandpass IIR	Bandpass
Bandstop FIR	<ul style="list-style-type: none"> • Bandstop • Bandstop (FIR)
Bandstop IIR	Bandstop

Note also that the transform options change depending on whether your original filter is FIR or IIR. Starting from an IIR filter, you can transform to IIR or FIR forms. With an IIR original filter, you are limited to IIR target filters.

After selecting your response type, use **Frequency point to transform** to specify the magnitude response point in your original filter to transfer to your target filter. Your target filter inherits the performance features of your original filter, such as passband ripple, while changing to the new response form.

For more information about transforming filters, refer to “Frequency Transformations for Real Filters” on page 3-95 and “Frequency Transformations for Complex Filters” on page 3-109.

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 3-87.

Transformed Filter Type

Select the magnitude response for the target filter from the list. The complete list of transformed filter types is:

- **Lowpass**
- **Lowpass (FIR)**
- **Highpass**
- **Highpass (FIR) narrowband**
- **Highpass (FIR) wideband**

- **Bandpass**
- **Bandstop**
- **Multiband**
- **Bandpass (complex)**
- **Bandstop (complex)**
- **Multiband (complex)**

Not all types of transformed filters are available for all filter types on the **Original filter types** list. You can transform bandpass filters only to bandpass filters. Or bandstop filters to bandstop filters. Or IIR filters to IIR filters.

For more information about transforming filters, refer to “Frequency Transformations for Real Filters” on page 3-95 and “Frequency Transformations for Complex Filters” on page 3-109.

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 3-87.

Transform Filters. To transform the magnitude response of your filter, use the **Transform Filter** option on the side bar.

1 Design or import your filter into FDATool.

2 Click **Transform Filter**, , on the side bar.

FDATool opens the **Transform Filter** panel in FDATool.

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

When you select the type, whether is **lowpass**, **highpass**, **bandpass**, or **bandstop**, FDATool recognizes whether your filter form is FIR or IIR. Using both your filter type selection and the filter form, FDATool adjusts the entries on the **Transformed filter type** list to show only those that apply to your original filter.

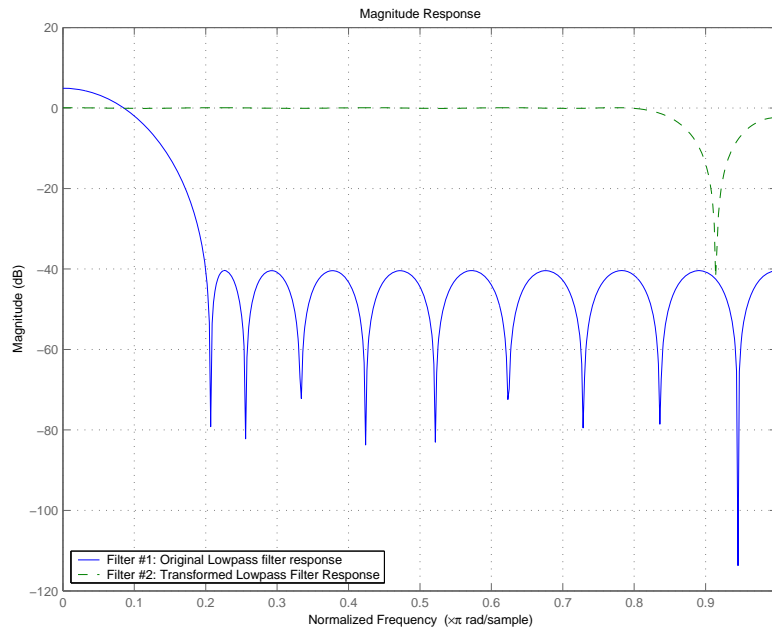
- 4 Enter the frequency point to transform value in **Frequency point to transform**. Notice that the value you enter must be in KHz; for example, enter 0.1 for 100 Hz or 1.5 for 1500 Hz.
- 5 From the **Transformed filter type** list, select the type of filter you want to transform to.

Your filter type selection changes the options here.

- When you pick a lowpass or highpass filter type, you enter one value in **Specify desired frequency location**.
- When you pick a bandpass or bandstop filter type, you enter two values — one in **Specify desired low frequency location** and one in **Specify desired high frequency location**. Your values define the edges of the passband or stopband.
- When you pick a multiband filter type, you enter values as elements in a vector in **Specify a vector or desired frequency locations** — one element for each desired location. Your values define the edges of the passbands and stopbands.

After you click **Transform Filter**, FDATool transforms your filter, displays the magnitude response of your new filter, and updates the **Current Filter Information** to show you that your filter has been transformed. In the filter information, the **Source** is **Transformed**.

For example, the figure shown here includes the magnitude response curves for two filter. The original filter is a lowpass filter with rolloff between 0.2 and 0.25. The transformed filter is a lowpass filter with rolloff region between 0.8 and 0.85.

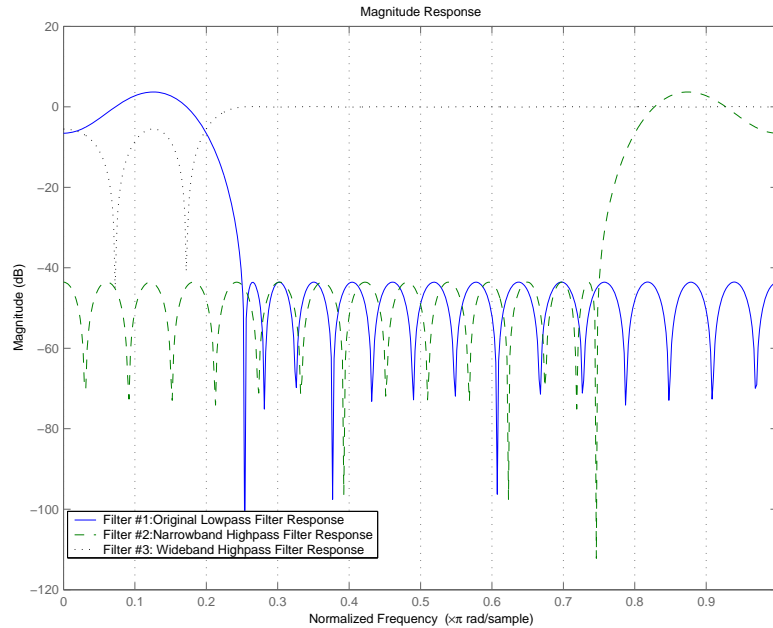


- To transform your lowpass filter to a highpass filter, select **Lowpass to Highpass**.

When you select **Lowpass to Highpass**, FDATool returns the dialog box shown here. More information about the **Select Transform...** dialog box follows the figure.



To demonstrate the effects of selecting **Narrowband Highpass** or **Wideband Highpass**, the next figure presents the magnitude response curves for a source lowpass filter after it is transformed to both narrow- and wideband highpass filters. For comparison, the response of the original filter appears as well.



For the narrowband case, the transformation algorithm essentially reverses the magnitude response, like reflecting the curve around the y -axis, then translating the curve to the right until the origin lies at 1 on the x -axis. After reflecting and translating, the passband at high frequencies is the reverse of the passband of the original filter at low frequencies with the same rolloff and ripple characteristics.

Design Multirate Filters in FDATool

- “Introduction” on page 3-70
- “Switch FDATool to Multirate Filter Design Mode” on page 3-70


- “Controls on the Multirate Design Panel” on page 3-71
- “Quantize Multirate Filters” on page 3-80
- “Export Individual Phase Coefficients of a Polyphase Filter to the Workspace” on page 3-82

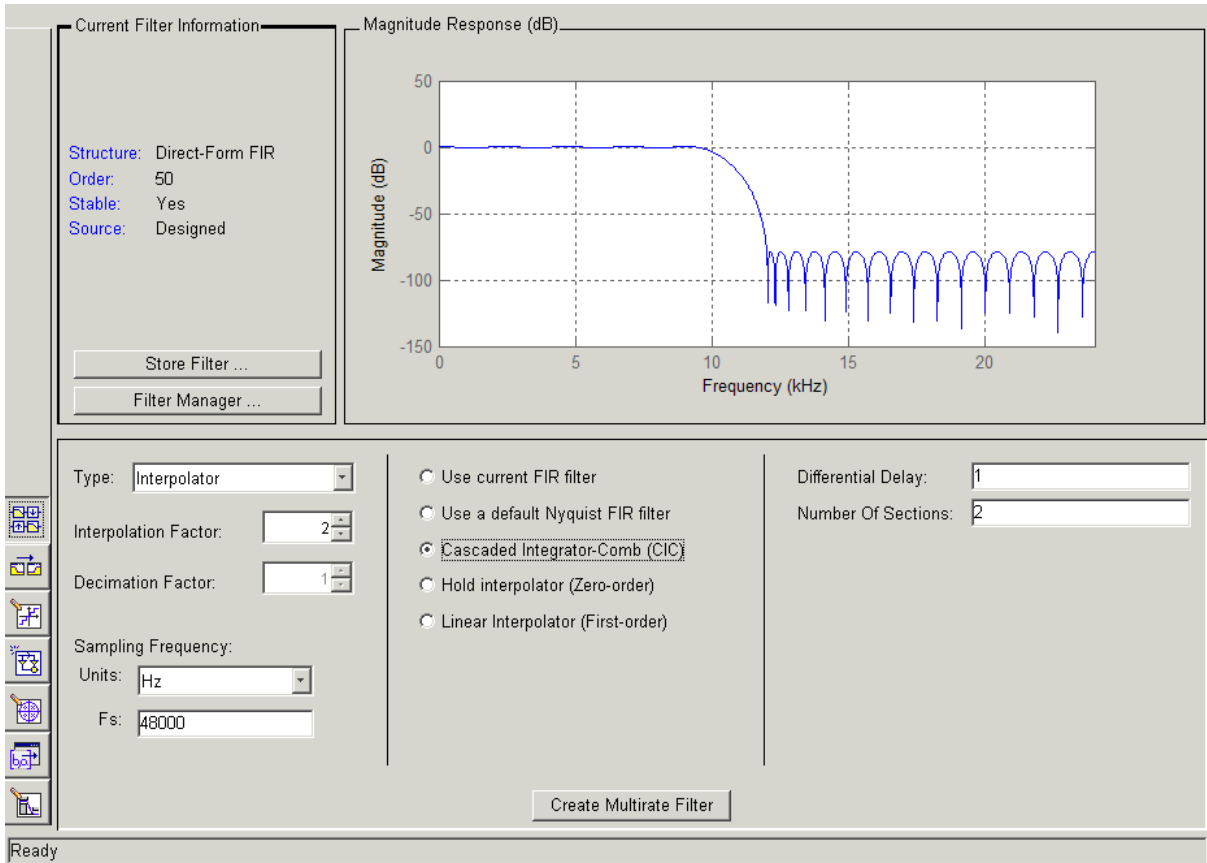
Introduction

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	<p>Specifies the type of multirate filter to design. Choose from Decimator, Interpolator, or Fractional-rate convertor.</p> <ul style="list-style-type: none"> • 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

Select and Configure Your Filter (Continued)

Option	Description
	the interpolation/decimation fractional rate to the lowest common denominator and issues a message in the status bar in FDATool. For example, if the interpolation factor is 6 and the decimation factor is 3, FDATool reduces 6/3 to 2/1 when you design the rate changer. But if the interpolation factor is 8 and the decimation factor is 3, FDATool designs the filter without change.
Interpolation Factor	Use the up-down control arrows to specify the amount of interpolation to apply to the signal. Factors range upwards from 2.
Decimation Factor	Use the up-down control arrows to specify the amount of decimation to apply to the signal. Factors range upwards from 2.
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.

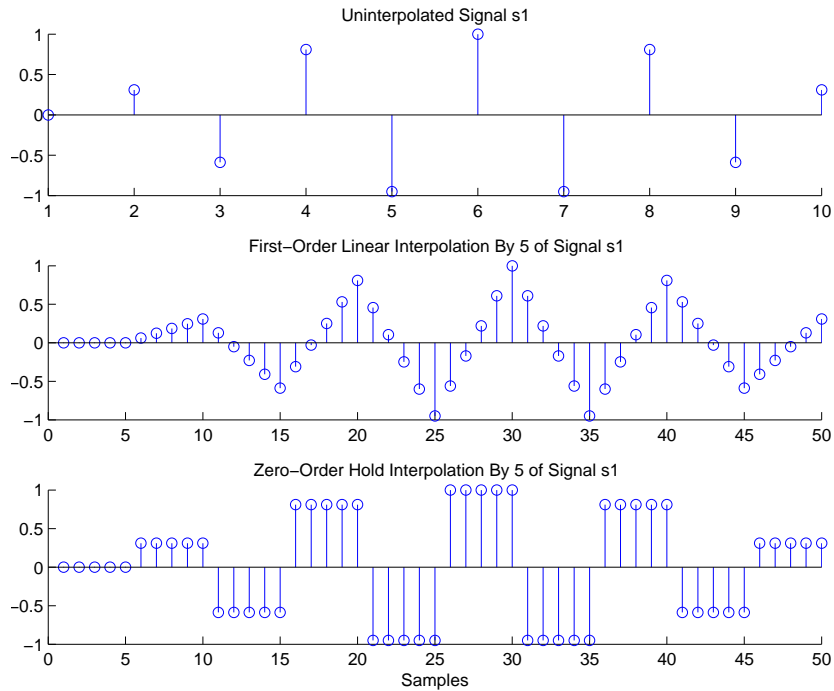
Design Your Filter (Continued)

Option	Description
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 s_1 in three forms:

- The top subplot in the figure presents signal s_1 without interpolation.
- The middle subplot shows signal s_1 interpolated by a linear interpolator with an interpolation factor of 5.
- The bottom subplot shows signal s_1 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:

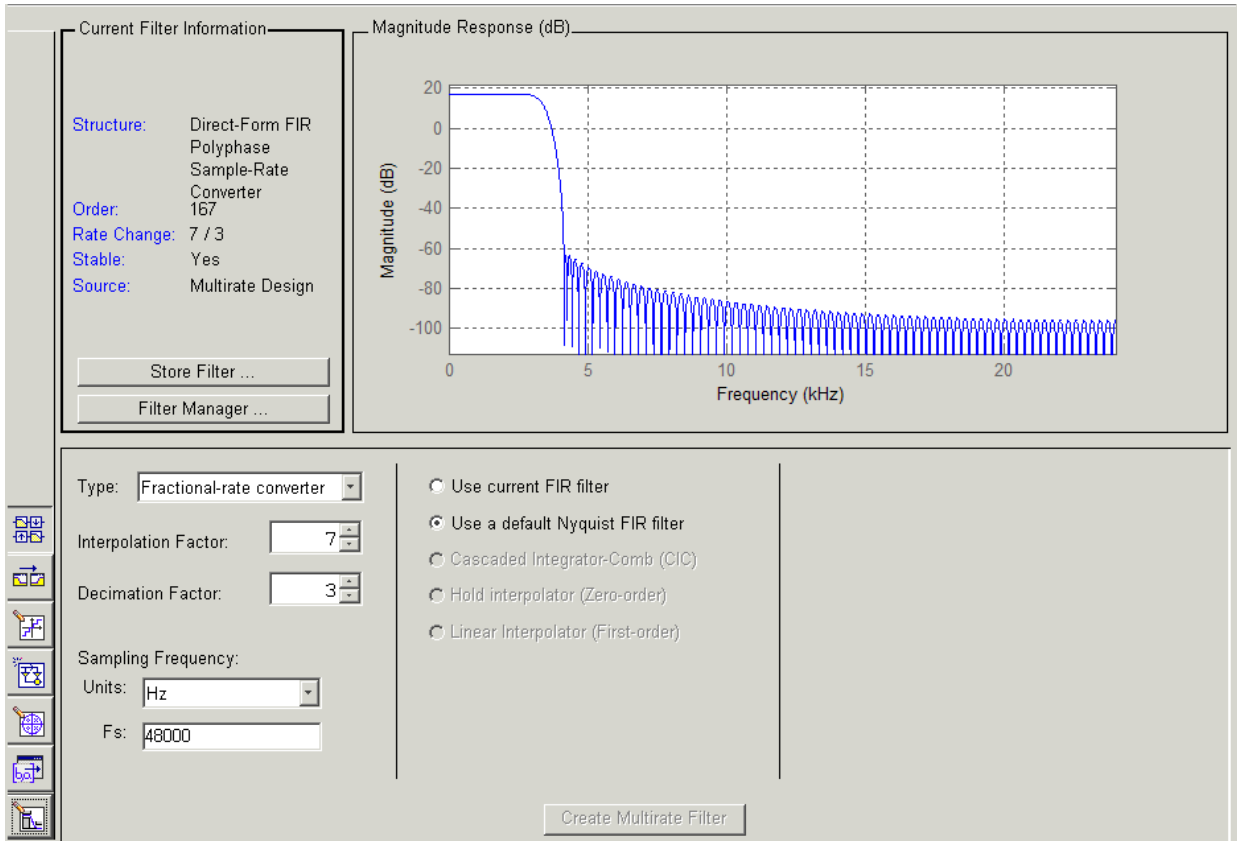
- Linear interpolator — `hm=mfilt.linearinterp(5)`
- Hold interpolator — `hm=mfilt.holdinterp(5)`

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.



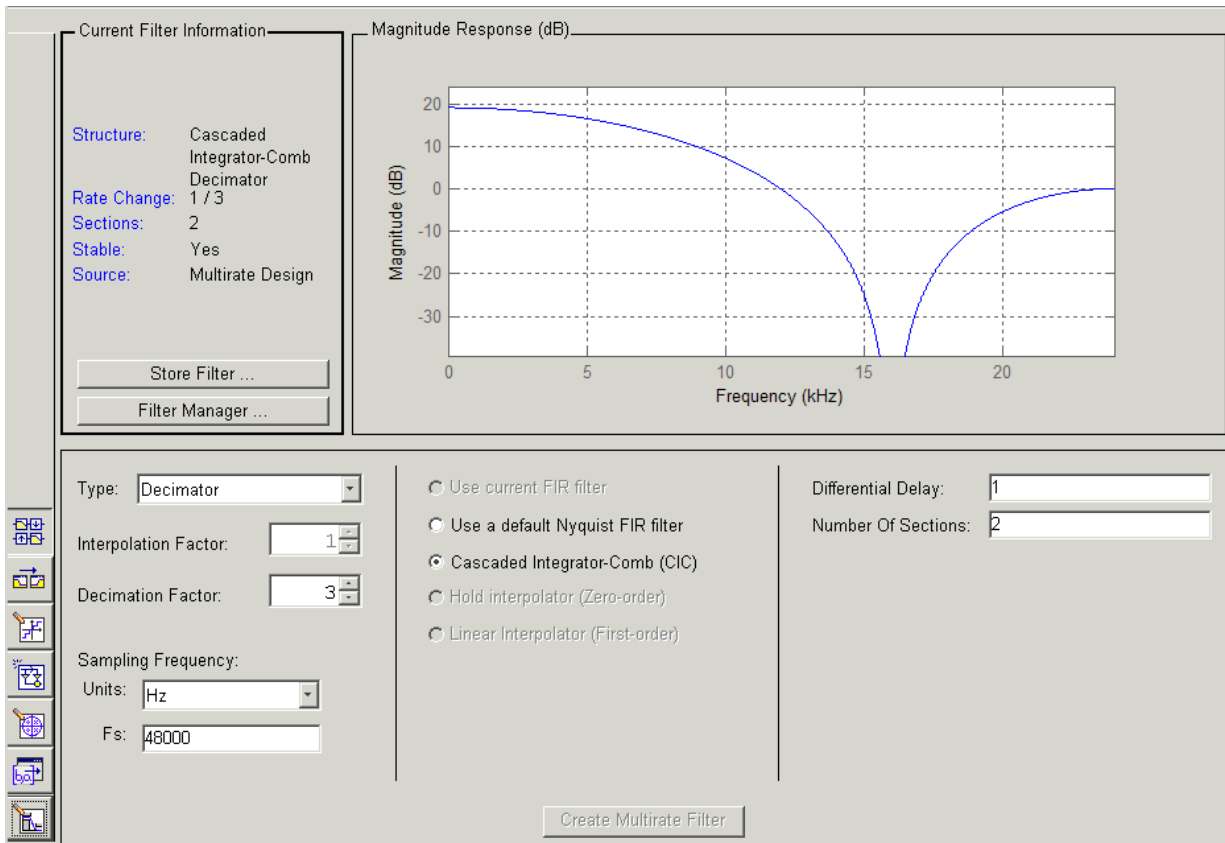
You can test the filter by exporting it to your workspace and using it to filter a signal. For information about exporting filters, refer to “Import and Export Quantized Filters” on page 3-53.

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 your current filter is a CIC filter, the options on the **Input/Output** and **Filter Internals** panes change to provide specific features for CIC filters.

Input/Output. The options that specify how your CIC filter uses input and output values are listed in the table below.

Option Name	Description
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, H_m 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, H_m . Each row of p consists of the coefficients of an individual phase subfilter. The first row contains 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 3-83
- “About the Realize Model Panel in FDATool” on page 3-83

Introduction

After you design or import a filter in FDATool, the realize model feature lets you create a Simulink subsystem block that implements your filter. The generated filter subsystem block uses either the Digital Filter block or the delay, gain, and sum blocks in Simulink. If you do not have a Simulink Fixed Point™ 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.

The screenshot shows a dialog box for realizing a filter model. It is divided into two main sections: 'Model' and 'Optimization'.
In the 'Model' section:
- 'Block name' is set to 'Filter'.
- 'Destination' is set to 'Current'.
- 'User Defined' is set to 'Untitled'.
- There are two checkboxes: 'Overwrite generated 'Filter' block' (unchecked) and 'Build model using basic elements' (checked).
In the 'Optimization' section:
- There are five checkboxes, all of which are unchecked: 'Optimize for zero gains', 'Optimize for unity gains', 'Optimize for negative gains', 'Optimize delay chains', and 'Optimize for unity scale values'.
At the bottom of the dialog:
- 'Input processing' is set to 'Columns as channels (frame based)'.
- A 'Realize Model' button is located on the right side.

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
<code>firdecim</code>	Decimators based on FIR filters
<code>firtdecim</code>	Decimators based on transposed FIR filters
<code>linearinterp</code>	Linear interpolators
<code>firinterp</code>	Interpolators based on FIR filters
<code>multirate polyphase</code>	Multirate filters
<code>holdinterp</code>	Interpolators that use the hold interpolation algorithm

Structure	Description
<code>dfilt.allpass</code>	Discrete-time filters with allpass structure
<code>dfilt.cascadeallpass</code>	
<code>dfilt.cascadewdfallpass</code>	
<code>mfilt.iirdecim</code>	Decimators based on IIR filters
<code>mfilt.iirwdfdecim</code>	
<code>mfilt.iirinterp</code>	Interpolators based on IIR filters
<code>mfilt.iirwdfinterp</code>	
<code>dfilt.wdfallpass</code>	

Digital Frequency Transformations

In this section...

“Details and Methodology” on page 3-87

“Frequency Transformations for Real Filters” on page 3-95

“Frequency Transformations for Complex Filters” on page 3-109

Details and Methodology

- “Overview of Transformations” on page 3-87
- “Select Features Subject to Transformation” on page 3-91
- “Mapping from Prototype Filter to Target Filter” on page 3-93
- “Summary of Frequency Transformations” on page 3-95

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 N th-order allpass mapping filter given by

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

$$\alpha_0 = 1$$

where

$H_o(z)$ — Transfer function of the prototype filter

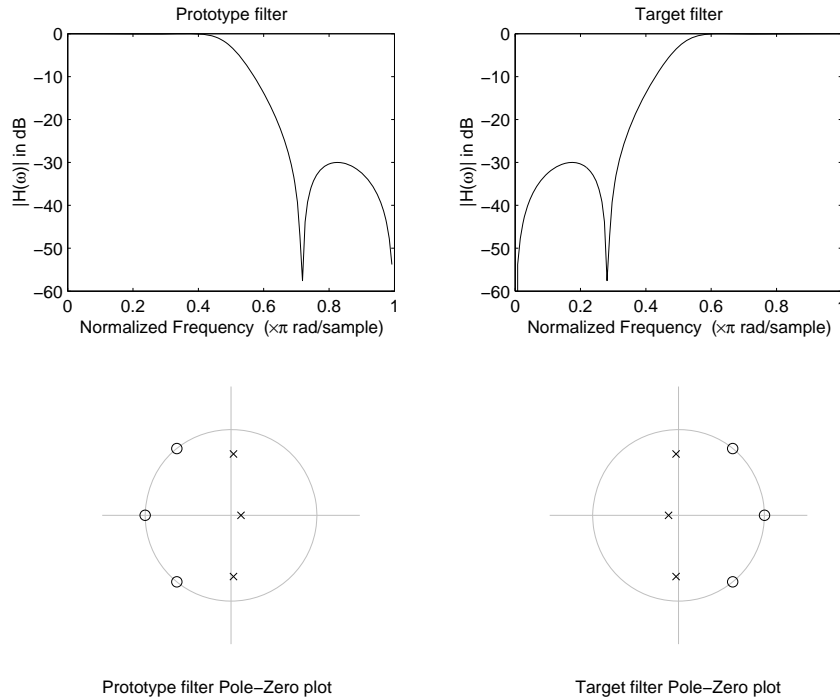
$H_A(z)$ — Transfer function of the allpass mapping filter

$H_T(z)$ — Transfer function of the target filter

Let's look at a simple example of the transformation given by

$$H_T(z) = H_o(-z)$$

The target filter has its poles and zeroes flipped across the origin of the real and imaginary axes. For the real filter prototype, it gives a mirror effect against 0.5, which means that lowpass $H_o(z)$ gives rise to a real highpass $H_T(z)$. This is shown in the following figure for the prototype filter designed as a third-order halfband elliptic filter.



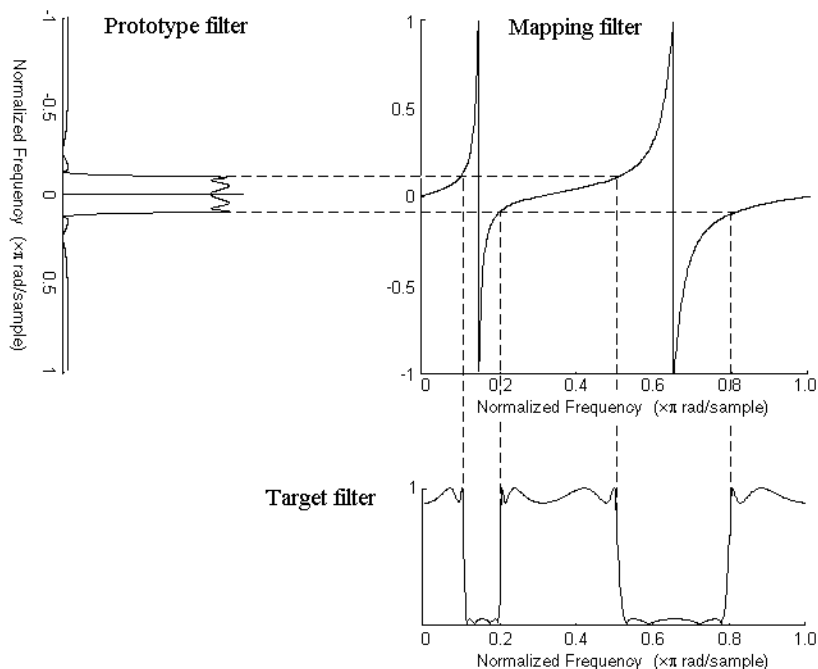
Example of a Simple Mirror Transformation

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:

$$H(\omega_{new}) = \omega_{old}$$

The graphical interpretation of the frequency transformation is shown in the figure below. The complex multiband transformation takes a real lowpass filter and converts it into a number of passbands around the unit circle.



Graphical Interpretation of the Mapping Process

Most of the frequency transformations are based on the second-order allpass mapping filter:

$$H_A(z) = \pm \frac{1 + \alpha_1 z^{-1} + \alpha_2 z^{-2}}{\alpha_2 + \alpha_1 z^{-1} + z^{-2}}$$

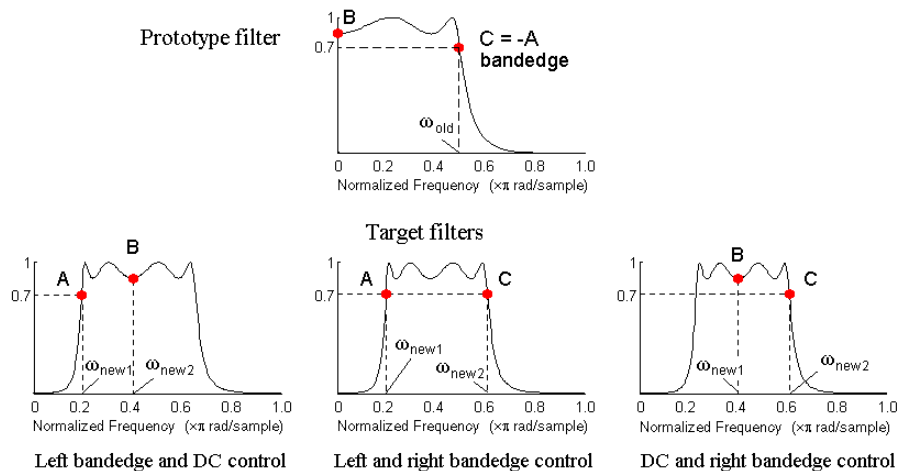
The two degrees of freedom provided by α_1 and α_2 choices are not fully used by the usual restrictive set of “flat-top” classical mappings like lowpass to bandpass. Instead, any two transfer function features can be migrated to (almost) any two other frequency locations if α_1 and α_2 are chosen so as to keep the poles of $H_A(z)$ strictly outside the unit circle (since $H_A(z)$ is substituted for z in the prototype transfer function). Moreover, as first pointed out by Constantinides, the selection of the outside sign influences whether the original feature at zero can be moved (the minus sign, a condition known

as “DC mobility”) or whether the Nyquist frequency can be migrated (the “Nyquist mobility” case arising when the leading sign is positive).

All the transformations forming the package are explained in the next sections of the tutorial. They are separated into those operating on real filters and those generating or working with complex filters. The choice of transformation ranges from standard Constantinides first and second-order ones [1][2] up to the real multiband filter by Mullis and Franchitti [3], and the complex multiband filter and real/complex multipoint ones by Krukowski, Cain and Kale [4].

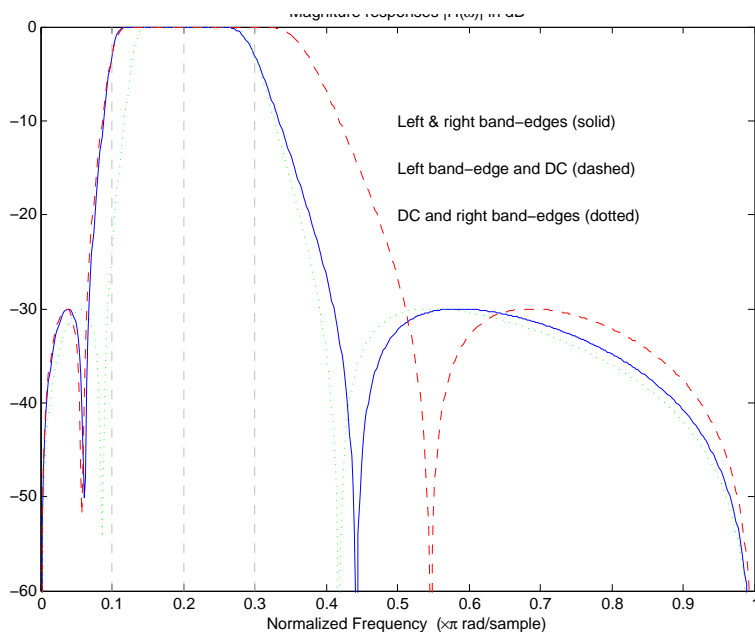
Select Features Subject to Transformation

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_A th-order allpass filter, $H_A(z)$. The general form of the allpass mapping filter is given in “Overview of Transformations” on page 3-87. The frequency mapping is a mathematical operation that replaces each delay of the prototype filter with an allpass filter. There are two ways of performing such mapping. The choice of the approach is dependent on how prototype and target filters are represented.

When the N th-order prototype filter is given with pole-zero form

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

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:

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

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

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

For each coefficient α_i and β_i of the prototype filter the N_A th-order polynomials must be convolved N times. Such approach may cause rounding errors for large prototype filters and/or high order mapping filters. In such a case the user should consider the alternative of doing the mapping using via poles and zeros.

Summary of Frequency Transformations

Advantages.

- Most frequency transformations are described by closed-form solutions or can be calculated from the set of linear equations.
- They give predictable and familiar results.
- Ripple heights from the prototype filter are preserved in the target filter.
- They are architecturally appealing for variable and adaptive filters.

Disadvantages.

- There are cases when using optimization methods to design the required filter gives better results.
- High-order transformations increase the dimensionality of the target filter, which may give expensive final results.
- Starting from fresh designs helps avoid locked-in compromises.

Frequency Transformations for Real Filters

- “Overview” on page 3-95
- “Real Frequency Shift” on page 3-96
- “Real Lowpass to Real Lowpass” on page 3-97
- “Real Lowpass to Real Highpass” on page 3-99
- “Real Lowpass to Real Bandpass” on page 3-101
- “Real Lowpass to Real Bandstop” on page 3-103
- “Real Lowpass to Real Multiband” on page 3-105
- “Real Lowpass to Real Multipoint” on page 3-107

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

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

where

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

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

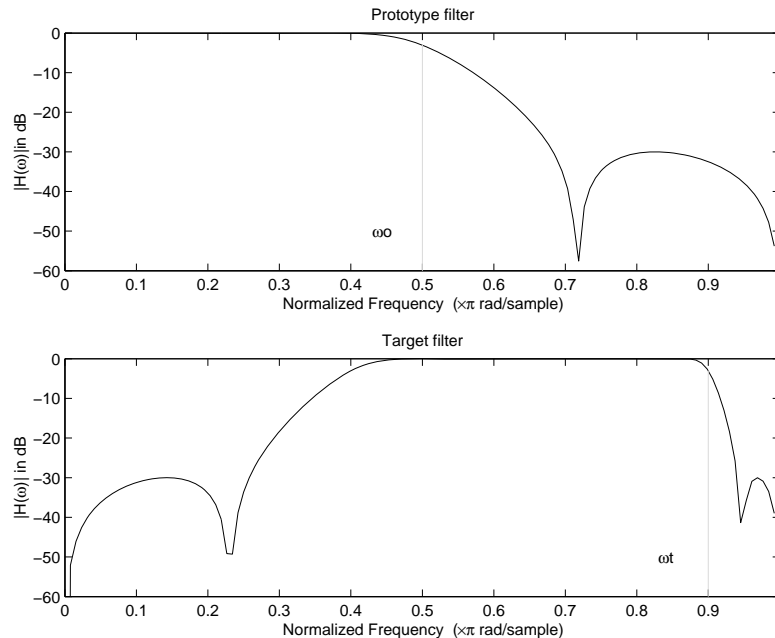
The following example shows how this transformation can be used to move the response of the prototype lowpass filter in either direction. Please note that because the target filter must also be real, the response of the target filter will inherently be disturbed at frequencies close to Nyquist and close to DC. Here is the MATLAB code for generating the example in the figure.

The prototype filter is a halfband elliptic, real, third-order lowpass filter:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

The example transformation moves the feature originally at 0.5 to 0.9:

```
[num,den] = iirshift(b, a, 0.5, 0.9);
```



Example of Real Frequency Shift Mapping

Real Lowpass to Real Lowpass

Real lowpass filter to real lowpass filter transformation uses a first-order allpass mapping filter. It performs an exact mapping of one feature of the frequency response into the new location keeping DC and Nyquist features fixed. As a real transformation, it works in a similar way for positive and negative frequencies. It is important to mention that using first-order mapping ensures that the order of the filter after the transformation is the same as it was originally.

$$H_A(z) = -\left(\frac{1 - \alpha z^{-1}}{\alpha - z^{-1}}\right)$$

with α given by

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

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

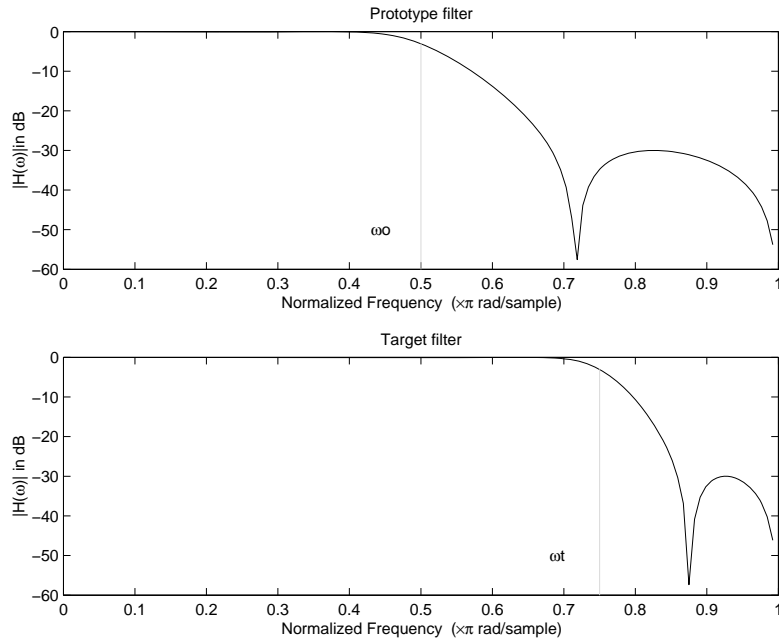
The example below shows how to modify the cutoff frequency of the prototype filter. The MATLAB code for this example is shown in the following figure.

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] = iir1p2lp(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

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

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

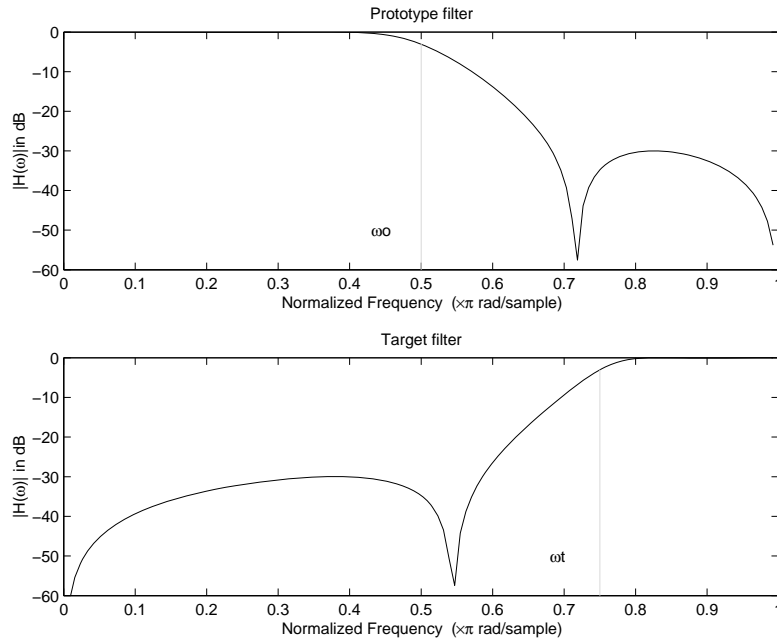
The example below shows how to convert the lowpass filter into a highpass filter with arbitrarily chosen cutoff frequency. In the MATLAB code below, the lowpass filter is converted into a highpass with cutoff frequency shifted from 0.5 to 0.75. Results are shown in the figure.

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] = iir1p2hp(b, a, 0.5, 0.75);
```



Example of Real Lowpass to Real Highpass Mapping

Real Lowpass to Real Bandpass

Real lowpass filter to real bandpass filter transformation uses a second-order allpass mapping filter. It performs an exact mapping of two features of the frequency response into their new location additionally moving a DC feature and keeping the Nyquist feature fixed. As a real transformation, it works in a similar way for positive and negative frequencies.

$$H_A(z) = - \left(\frac{1 - \beta(1 + \alpha)z^{-1} - \alpha z^{-2}}{\alpha - \beta(1 + \alpha)z^{-1} + z^{-2}} \right)$$

with α and β given by

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

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

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

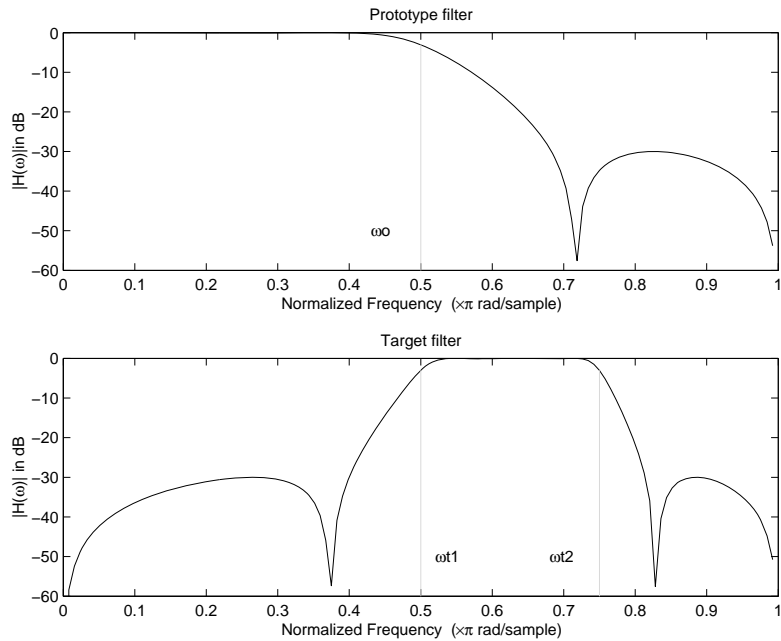
The example below shows how to move the response of the prototype lowpass filter in either direction. Please note that because the target filter must also be real, the response of the target filter will inherently be disturbed at frequencies close to Nyquist and close to DC. Here is the MATLAB code for 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] = iir1p2bp(b, a, 0.5, [0.5, 0.75]);
```

Example of Real Lowpass to Real Bandpass Mapping

Real Lowpass to Real Bandstop

Real lowpass filter to real bandstop filter transformation uses a second-order allpass mapping filter. It performs an exact mapping of two features of the frequency response into their new location additionally moving a Nyquist feature and keeping the DC feature fixed. This effectively creates a stopband between the selected frequency locations in the target filter. As a real transformation, it works in a similar way for positive and negative frequencies.

$$H_A(z) = \frac{1 - \beta(1 + \alpha)z^{-1} + \alpha z^{-2}}{\alpha - \beta(1 + \alpha)z^{-1} + z^{-2}}$$

with α and β given by

$$\alpha = \frac{\cos \frac{\pi}{4} (2\omega_{old} + \omega_{new,2} - \omega_{new,1})}{\cos \frac{\pi}{4} (2\omega_{old} - \omega_{new,2} + \omega_{new,1})}$$
$$\beta = \cos \frac{\pi}{2} (\omega_{new,1} + \omega_{new,2})$$

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

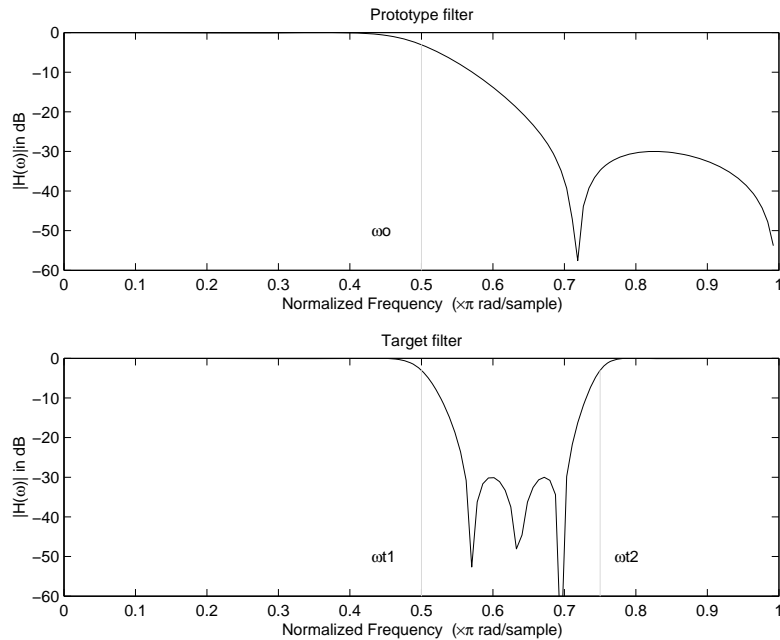
The following example shows how this transformation can be used to convert the prototype lowpass filter with cutoff frequency at 0.5 into a real bandstop filter with the same passband and stopband ripple structure and stopband positioned between 0.5 and 0.75. Here is the MATLAB code for generating the 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] = iir1p2bs(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.

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

$$\alpha_0 = 1$$

The coefficients a are given by

$$\begin{cases} \alpha_0 = 1 & k = 1, \dots, N \\ \alpha_k = -S \frac{\sin \frac{\pi}{2} (N\omega_{new} + (-1)^k \omega_{old})}{\sin \frac{\pi}{2} ((N - 2k)\omega_{new} + (-1)^k \omega_{old})} \end{cases}$$

where

$\omega_{old,k}$ — Frequency location of the first feature in the prototype filter

$\omega_{new,k}$ — Position of the feature originally at $\omega_{old,k}$ in the target filter

The mobility factor, S , specifies the mobility or either DC or Nyquist feature:

$$S = \begin{cases} 1 & \text{Nyquist} \\ -1 & \text{DC} \end{cases}$$

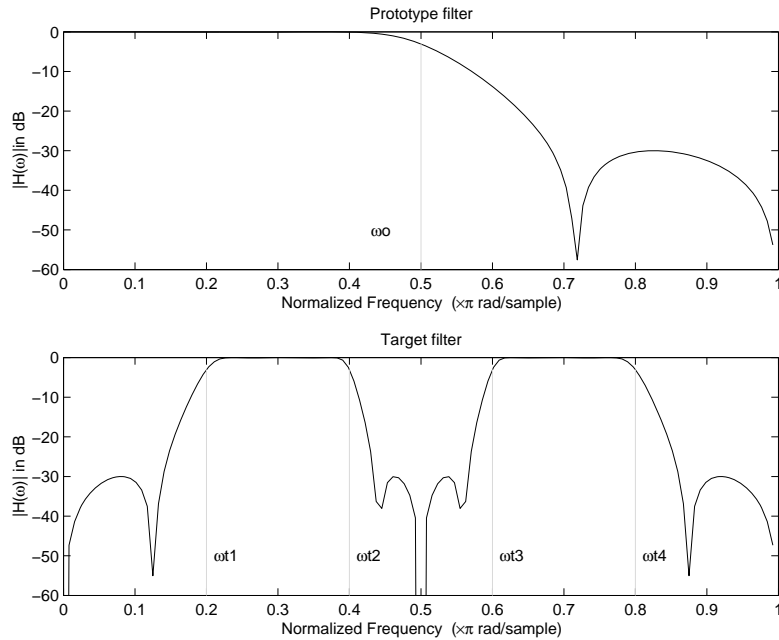
The example below shows how this transformation can be used to convert the prototype lowpass filter with cutoff frequency at 0.5 into a filter having a number of bands positioned at arbitrary edge frequencies 1/5, 2/5, 3/5 and 4/5. Parameter S was such that there is a passband at DC. Here is the MATLAB 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] = iir1p2mb(b, a, 0.5, [0.2, 0.4, 0.6, 0.8], 'pass');
```



Example of Real Lowpass to Real Multipoint Mapping

Real Lowpass to Real Multipoint

This high-order frequency transformation performs an exact mapping of a number of selected features of the prototype filter frequency response to their new locations in the target filter. The mapping filter is given by the general IIR polynomial form of the transfer function as given below.

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

$$\alpha_0 = 1$$

For the N th-order multipoint frequency transformation the coefficients a are

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

where

$\omega_{old,k}$ — Frequency location of the first feature in the prototype filter

$\omega_{new,k}$ — Position of the feature originally at $\omega_{old,k}$ in the target filter

The mobility factor, S , specifies the mobility of either DC or Nyquist feature:

$$S = \begin{cases} 1 & \text{Nyquist} \\ -1 & \text{DC} \end{cases}$$

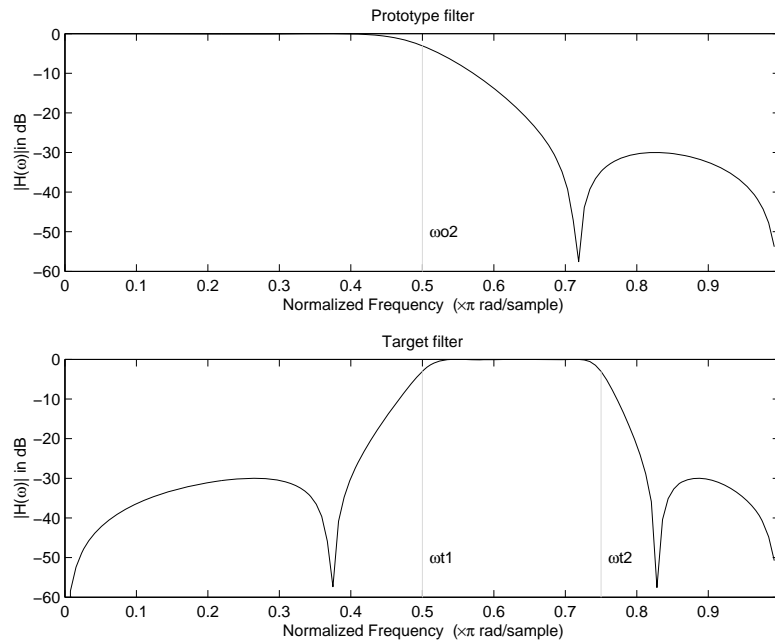
The example below shows how this transformation can be used to move features of the prototype lowpass filter originally at -0.5 and 0.5 to their new locations at 0.5 and 0.75, effectively changing a position of the filter passband. Here is the MATLAB code for generating the figure.

The prototype filter is a halfband elliptic, real, third-order lowpass filter:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

The example transformation creates a passband from 0.5 to 0.75:

```
[num,den] = iir1p2xn(b, a, [-0.5, 0.5], [0.5, 0.75], 'pass');
```



Example of Real Lowpass to Real Multipoint Mapping

Frequency Transformations for Complex Filters

- “Overview” on page 3-109
- “Complex Frequency Shift” on page 3-110
- “Real Lowpass to Complex Bandpass” on page 3-111
- “Real Lowpass to Complex Bandstop” on page 3-113
- “Real Lowpass to Complex Multiband” on page 3-115
- “Real Lowpass to Complex Multipoint” on page 3-117
- “Complex Bandpass to Complex Bandpass” on page 3-119

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

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

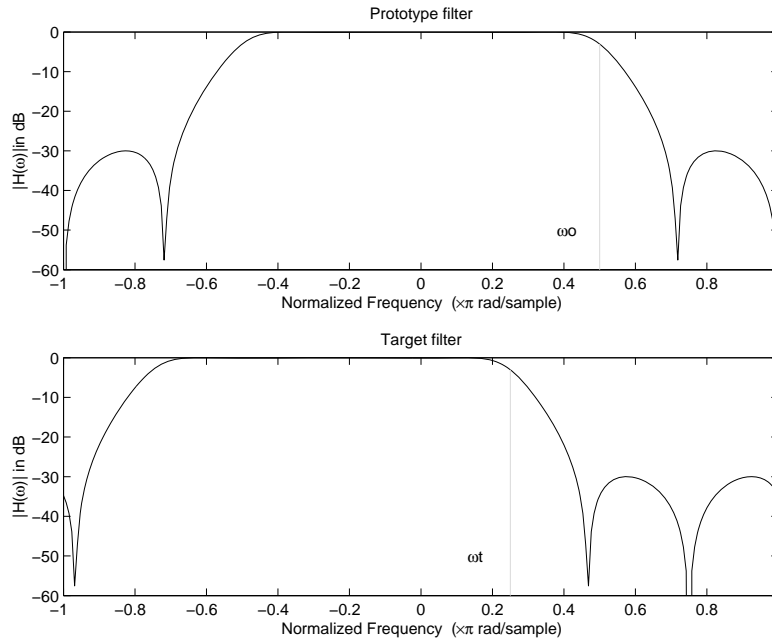
The example below shows how to apply this transformation to rotate the response of the prototype lowpass filter in either direction. Please note that because the transformation can be achieved by a simple phase shift operator, all features of the prototype filter will be moved by the same amount. Here is the MATLAB code for generating the example in the figure.

The prototype filter is a halfband elliptic, real, third-order lowpass filter:


```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

The example transformation moves the feature originally at 0.5 to 0.3:

```
[num,den] = iirshiftfc(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 a and β are given by

$$\alpha = \frac{\sin \frac{\pi}{4} (2\omega_{old} - \omega_{new,2} + \omega_{new,1})}{\sin \pi (2\omega_{old} + \omega_{new,2} - \omega_{new,1})}$$
$$\beta = e^{-j\pi(\omega_{new} - \omega_{old})}$$

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

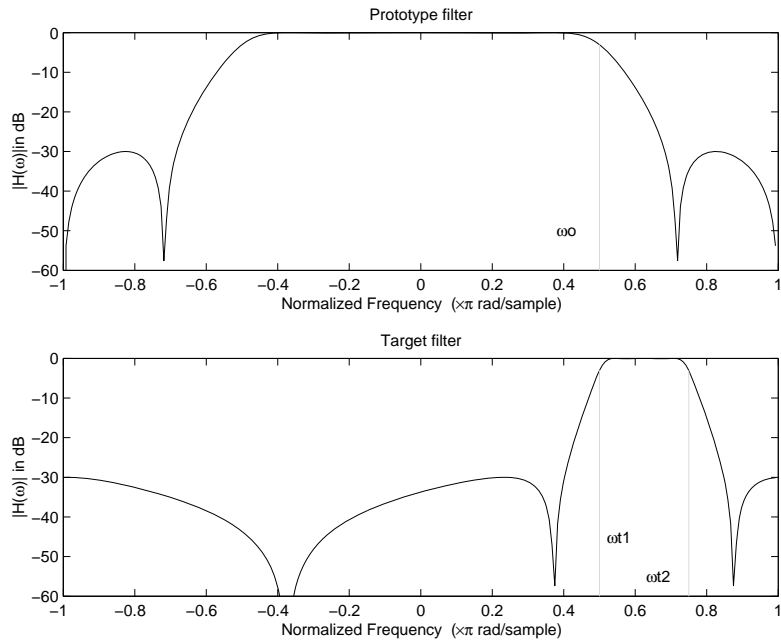
The following example shows the use of such a transformation for converting a real halfband lowpass filter into a complex bandpass filter with band edges at 0.5 and 0.75. Here is the MATLAB code for generating the example in the figure.

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] = iir1p2bpc(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

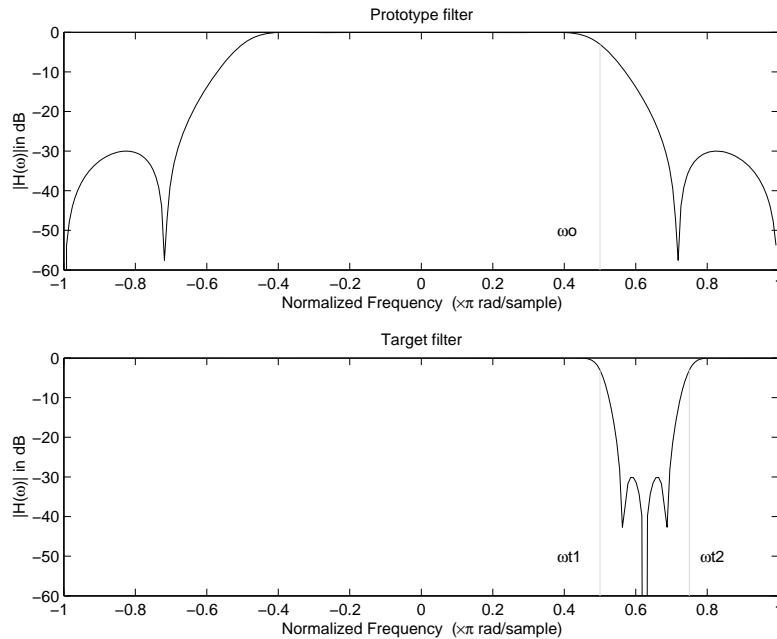
The example below shows the use of such a transformation for converting a real halfband lowpass filter into a complex bandstop filter with band edges at 0.5 and 0.75. Here is the MATLAB code for generating the example in the figure.

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] = iir1p2bsc(b, a, 0.5, [0.5 0.75]);
```



Example of Real Lowpass to Complex Bandstop Mapping

Real Lowpass to Complex Multiband

This high-order transformation performs an exact mapping of one selected feature of the prototype filter frequency response into a number of new locations in the target filter. Its most common use is to convert a real lowpass with predefined passband and stopband ripples into a multiband filter with arbitrary band edges. The order of the mapping filter must be even, which corresponds to an even number of band edges in the target filter. The N th-order complex allpass mapping filter is given by the following general transfer function form:

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

$$\alpha_0 = 1$$

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

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

where

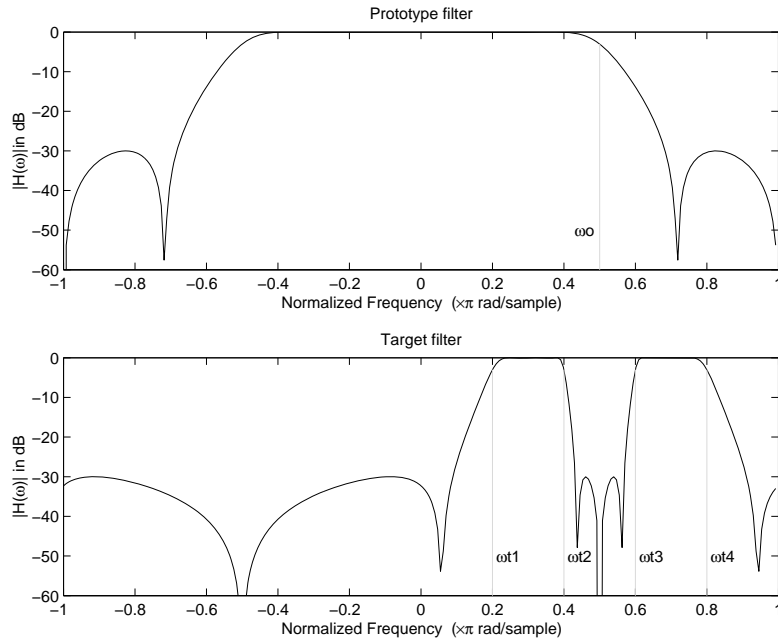
ω_{old} — Frequency location of the selected feature in the prototype filter

$\omega_{new,i}$ — Position of features originally at $\pm\omega_{old}$ in the target filter

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

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

The example shows the use of such a transformation for converting a prototype real lowpass filter with the cutoff frequency at 0.5 into a multiband complex filter with band edges at 0.2, 0.4, 0.6 and 0.8, creating two passbands around the unit circle. Here is the MATLAB code for generating the figure.



Example of Real Lowpass to Complex Multiband Mapping

The prototype filter is a halfband elliptic, real, third-order lowpass filter:

```
[b, a] = ellip(3, 0.1, 30, 0.409);
```

The example transformation creates two complex passbands:

```
[num,den] = iir1p2mbc(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.

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

$$\alpha_0 = 1$$

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

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

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

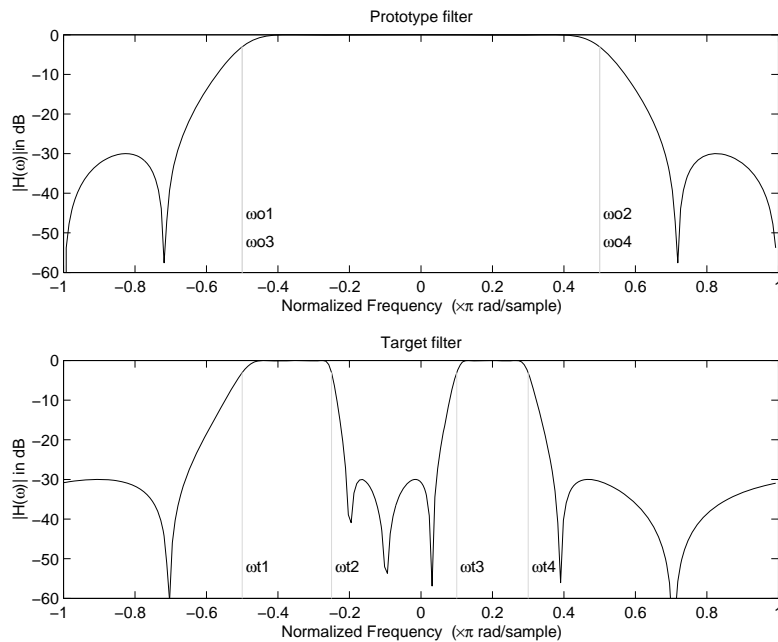
from 0.5 to new locations at -0.25 and 0.3. This creates two nonsymmetric passbands around the unit circle, creating a complex filter. Here is the MATLAB code for generating the figure.

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] = iir1p2xc(b,a,0.5*[-1,1,-1,1], [-0.5,-0.25,0.1,0.3]);
```



Example of Real Lowpass to Complex Multipoint Mapping

Complex Bandpass to Complex Bandpass

This first-order transformation performs an exact mapping of two selected features of the prototype filter frequency response into two new locations in the target filter. Its most common use is to adjust the edges of the complex bandpass filter.

$$H_A(z) = \frac{\alpha(\gamma - \beta z^{-1})}{z^{-1} - \beta\gamma}$$

with α and β are given by

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

$$\alpha = e^{-j\pi(\omega_{old,2} - \omega_{old,1})}$$

$$\gamma = e^{-j\pi(\omega_{new,2} - \omega_{new,1})}$$

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

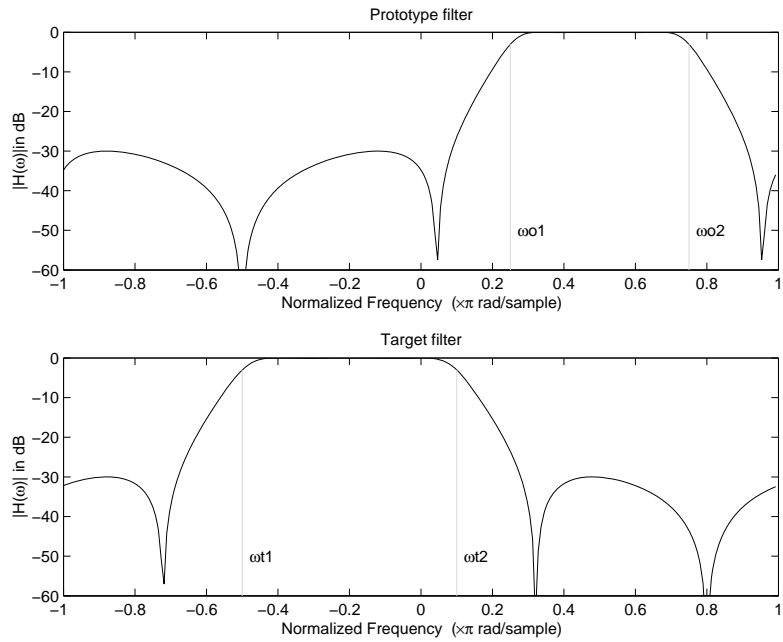
The following example shows how this transformation can be used to modify the position of the passband of the prototype filter, either real or complex. In the example below the prototype filter passband spanned from 0.5 to 0.75. It was converted to having a passband between -0.5 and 0.1. Here is the 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]);
```



Example of Complex Bandpass to Complex Bandpass Mapping

Digital Filter Design Block

In this section...
“Overview of the Digital Filter Design Block” on page 3-122
“Select a Filter Design Block” on page 3-123
“Create a Lowpass Filter in Simulink” on page 3-125
“Create a Highpass Filter in Simulink” on page 3-126
“Filter High-Frequency Noise in Simulink” on page 3-128

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 Code Generation” on page 9-2.

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” in the Signal Processing Toolbox documentation. To learn how to import and export your filter designs, see “Import and Export Quantized Filters” on page 3-53.

Note You can use the Digital Filter Design block to design and implement a filter. To implement a pre-designed filter, use the Digital Filter block. Both blocks 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 3-123.

Select a Filter Design Block

This section explains the similarities and differences between the Digital Filter Design and Filter Realization Wizard blocks.

Note In addition to these two blocks, you can also design filters with blocks from the `dspfdesign` library. Blocks from the `dspfdesign` library support the same set of filter structures as the Digital Filter Design and Filter Realization Wizard blocks. However, in some cases, the `dspfdesign` library blocks will provide you with more accurate numerical results.

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. For more information, see Chapter 9, “Code Generation”.
- 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 3-134 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 3-126, you use a Digital Filter Design block to create a highpass filter. For more information about implementing a pre-designed filter, see “Digital Filter Block” on page 3-147.

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 3-125 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 3-125. 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 3-128, 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 Block” on page 3-147.

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 3-126 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.

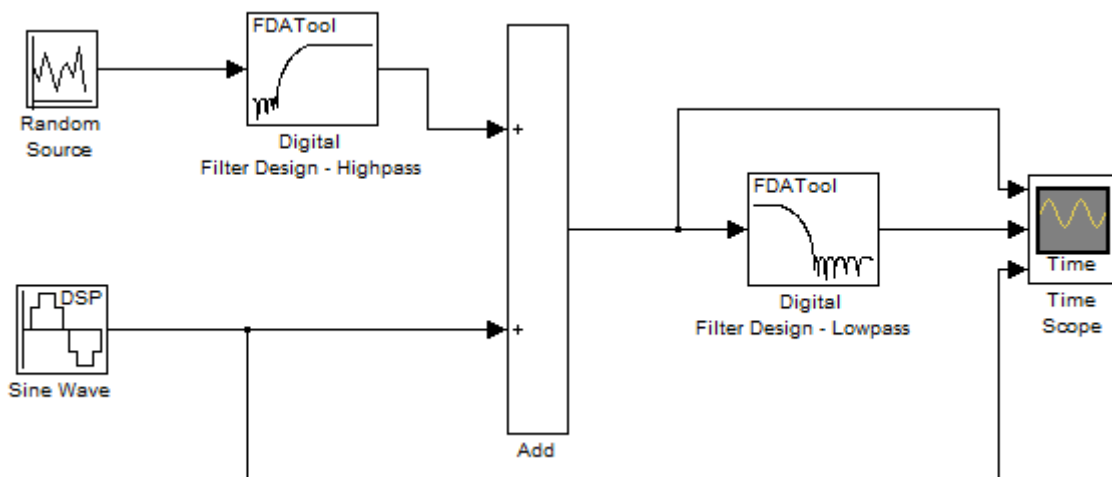
Block	Library	Quantity
Add	Simulink Math Operations library	1
Random Source	Sources	1
Sine Wave	Sources	1
Time Scope	Sinks	1

- 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	<ul style="list-style-type: none"> • Icon shape = rectangular • List of signs = ++
Random Source	<ul style="list-style-type: none"> • Source type = Uniform • Minimum = 0 • Maximum = 4 • Sample mode = Discrete • Sample time = 1/1000 • Samples per frame = 50
Sine Wave	<ul style="list-style-type: none"> • Frequency (Hz) = 75 • Sample time = 1/1000 • Samples per frame = 50
Time Scope	<ul style="list-style-type: none"> • File > Number of Input Ports > 3 • File > Configuration ... <ul style="list-style-type: none"> - 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 **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 **Start**.

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

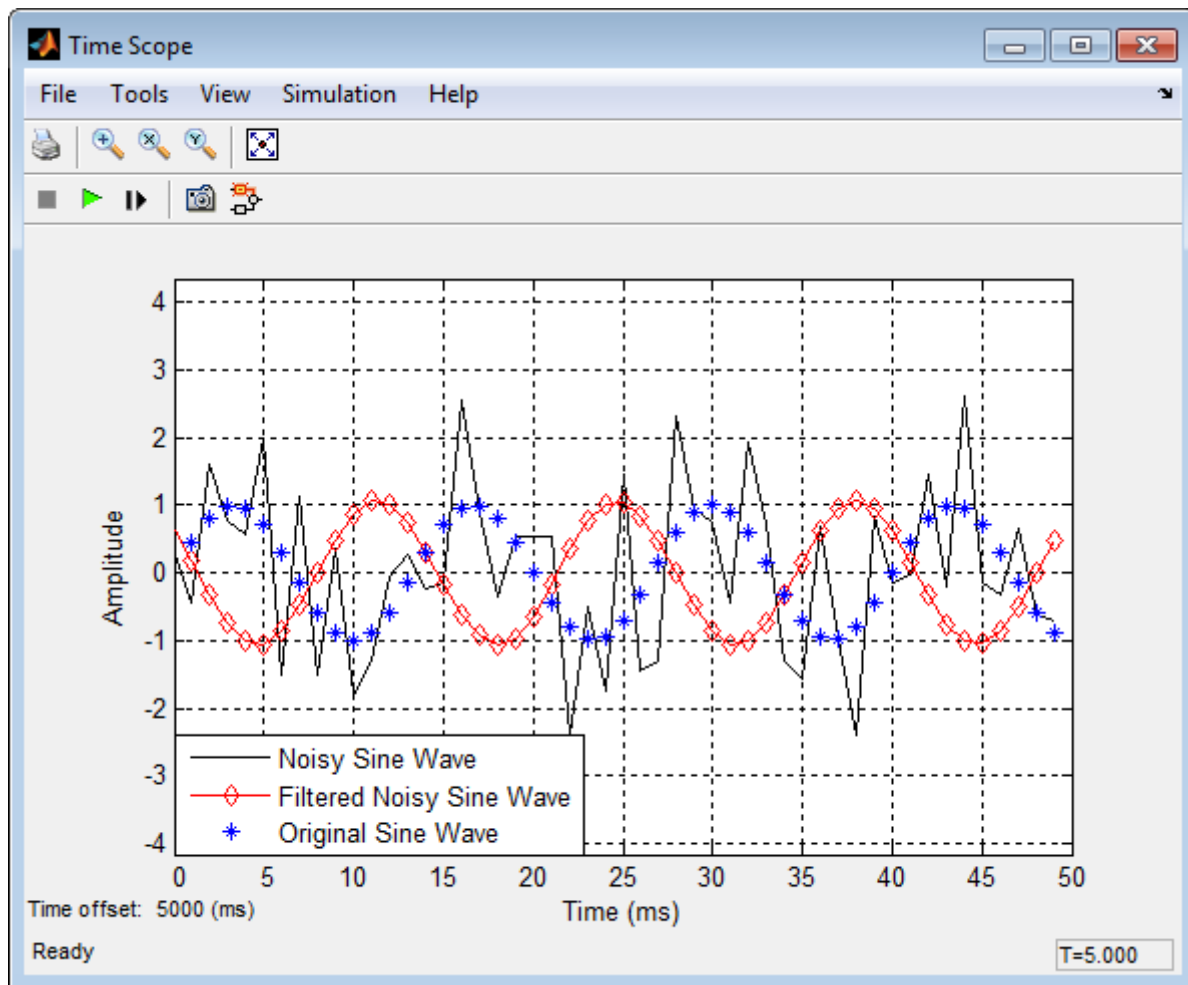
- Sine Wave = Original Sine Wave

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 3-134. To learn how to save your filter designs,

see “Saving and Opening Filter Design Sessions” in the Signal Processing Toolbox documentation. To learn how to import and export your filter designs, see “Import and Export Quantized Filters” on page 3-53 in the DSP System Toolbox documentation.

Filter Realization Wizard

In this section...
“Overview of the Filter Realization Wizard” on page 3-134
“Design and Implement a Fixed-Point Filter in Simulink” on page 3-134
“Set the Filter Structure and Number of Filter Sections” on page 3-143
“Optimize the Filter Structure” on page 3-144

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 3-123.

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 3-135
- “Part 2 — Create a Fixed-Point Filter with the Filter Realization Wizard” on page 3-136
- “Part 3 — Build a Model to Filter a Signal” on page 3-141
- “Part 4 — Examine Filtering Results” on page 3-142

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

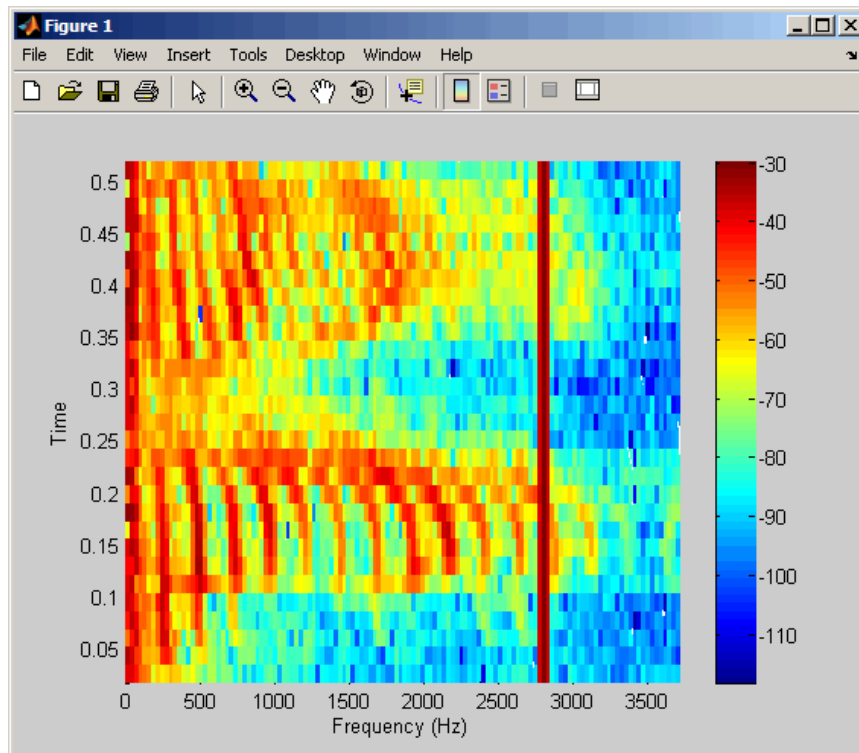
```
soundsc(u,Fs)
```

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 $3 \cdot F_s / 8$.

Part 2 – Create a Fixed-Point Filter with the Filter Realization Wizard

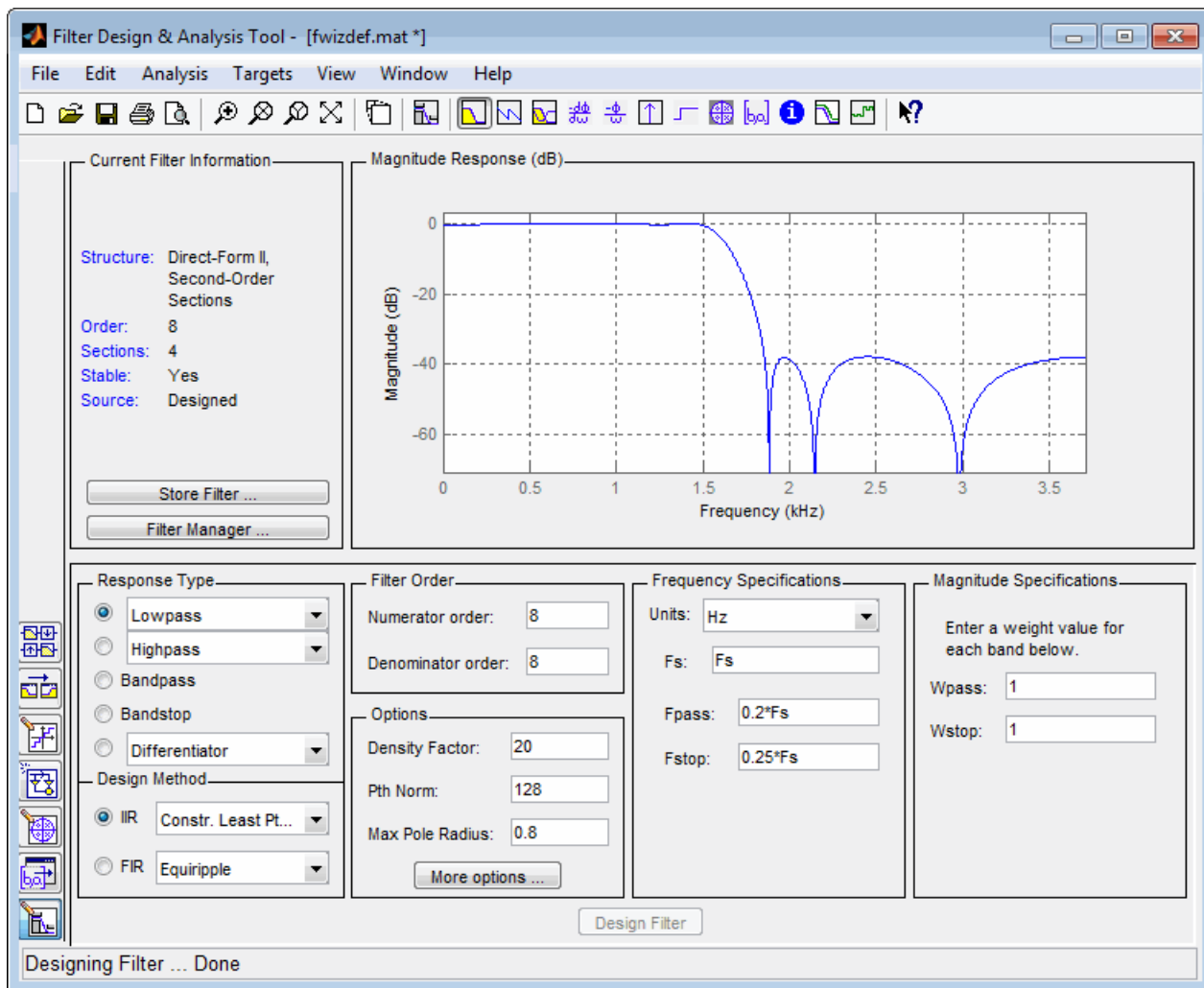
Next you will create a fixed-point filter using the Filter Realization Wizard. You will create a filter that reduces the effects of the noise on the signal.


- 6 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.

- 7** Double-click the Filter Realization Wizard block in your model. The **Realize Model** panel of the Filter Design and Analysis Tool (FDATool) appears.
- 8** Click the Design Filter button () on the bottom left of FDATool. This brings forward the **Design filter** panel of the tool.
- 9** 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 $0.2 \cdot F_s$
 - Set **Fstop** to $0.25 \cdot F_s$
 - Set **Max pole radius** to 0.8
 - Click the **Design Filter** button

The **Design filter** panel should now appear as follows.



10 Click the **Set quantization parameters** button on the bottom left of FDATool (). This brings forward the **Set quantization parameters** panel of the tool.

11 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.

The screenshot displays the 'Filter Design & Analysis Tool' window. The 'Current Filter Information' panel on the left shows the following details:


- Structure: Direct-Form II, Second-Order Sections
- Order: 8
- Sections: 4
- Stable: Yes
- Source: Designed (quantized)

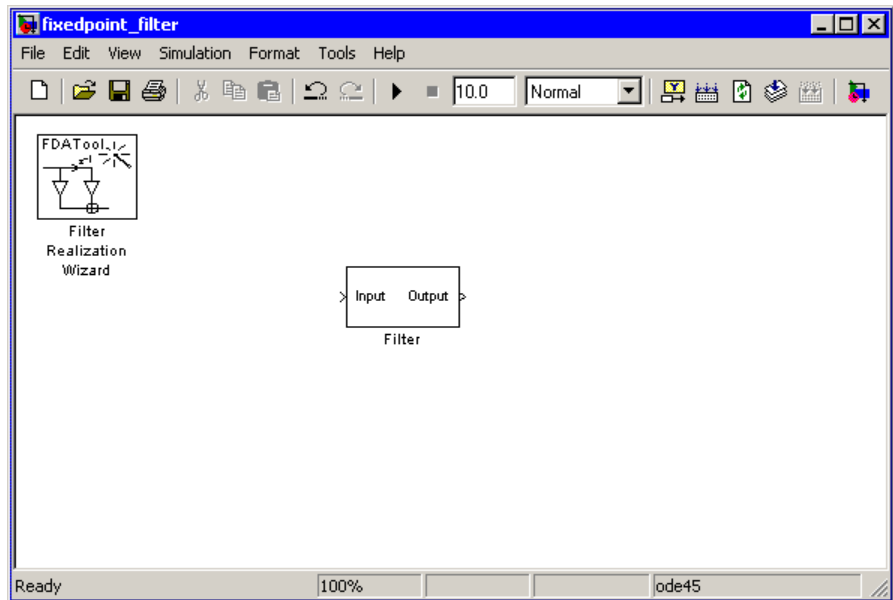
The 'Magnitude Response (dB)' plot shows the magnitude response in dB versus Frequency (kHz). The plot compares the 'Quantized' response (solid blue line) with the 'Reference' response (dashed blue line). The quantized response shows a slight deviation from the reference response, particularly in the passband and stopband regions.

The 'Filter arithmetic' panel is set to 'Fixed-point'. The 'Coefficients' tab is active, showing the following settings:

- Filter arithmetic: Fixed-point
- Coefficient word length: 16
- Best-precision fraction lengths
- Use unsigned representation
- Numerator frac. length: 14
- Scale Values frac. length: 18
- Numerator range (+/-): 1
- Scale Values range (+/-): 1
- Denominator frac. length: 14
- Denominator range (+/-): 1

The 'Apply' button is visible at the bottom of the 'Coefficients' tab. The status bar at the bottom of the window indicates 'Quantizing Filter ... done'.

- 12 Click the Realize Model button on the left side of FDATool (). This brings forward the **Realize Model** panel of the tool.
- 13 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.



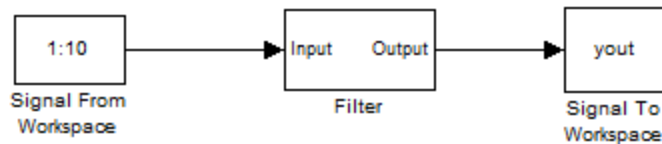
Note You do not have to keep the Filter Realization Wizard block in the same model as the generated Filter block. However, for this tutorial, we will keep the blocks in the same model.

- 14 Double-click the Filter subsystem block in your model to view the filter implementation.

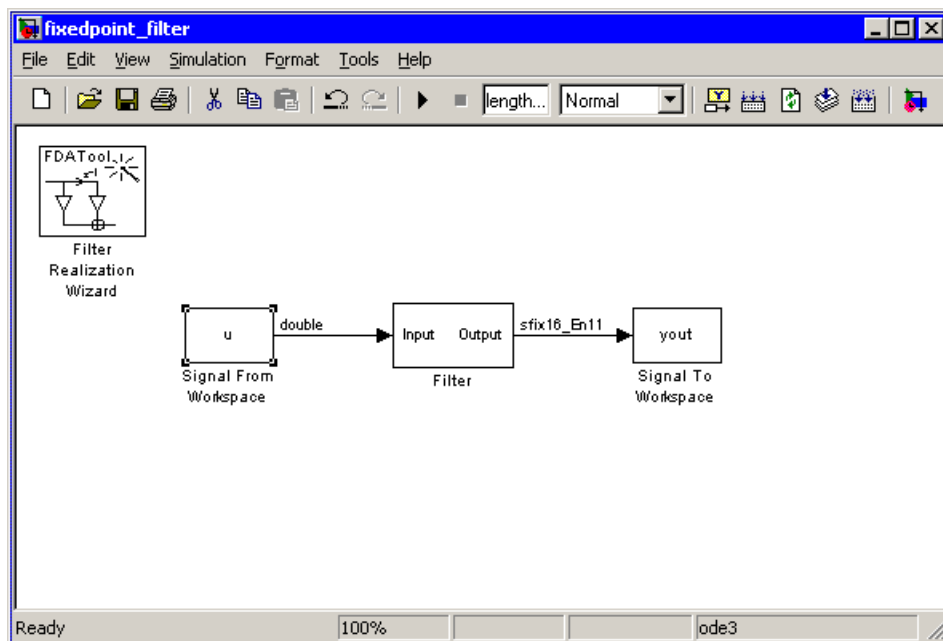
Part 3 – Build a Model to Filter a Signal

In this section of the tutorial, you will filter noise from a signal in your Simulink model.

- 15 Connect a Signal From Workspace block from the Sources library to the input port of your filter block.
- 16 Connect a Signal To Workspace block from the Sinks library to the output port of your filter block. Your blocks should now be connected as follows.



- 17 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.
- 18 Open the **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.
- 19 Run the model.
- 20 From the **Format** menu of the model, select **Port/Signal Displays > 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.

21 Type

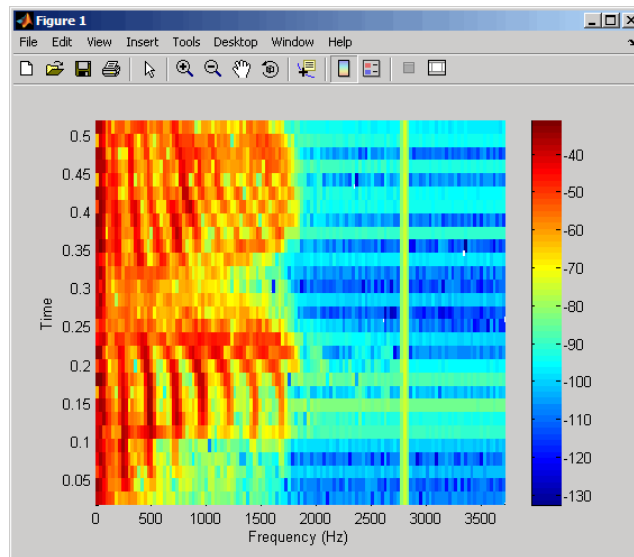
```
soundsc(yout,Fs)
```

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.

22 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.

Note You might not be able to directly access some of the supported structures through the **Convert Structure** dialog of FDATool. However, you *can* access all of the structures by creating a `dfilt` filter object with the desired structure, and then importing the filter into FDATool. To learn more about the **Import Filter** panel, see “Importing a Filter Design” in the Signal Processing Toolbox documentation.

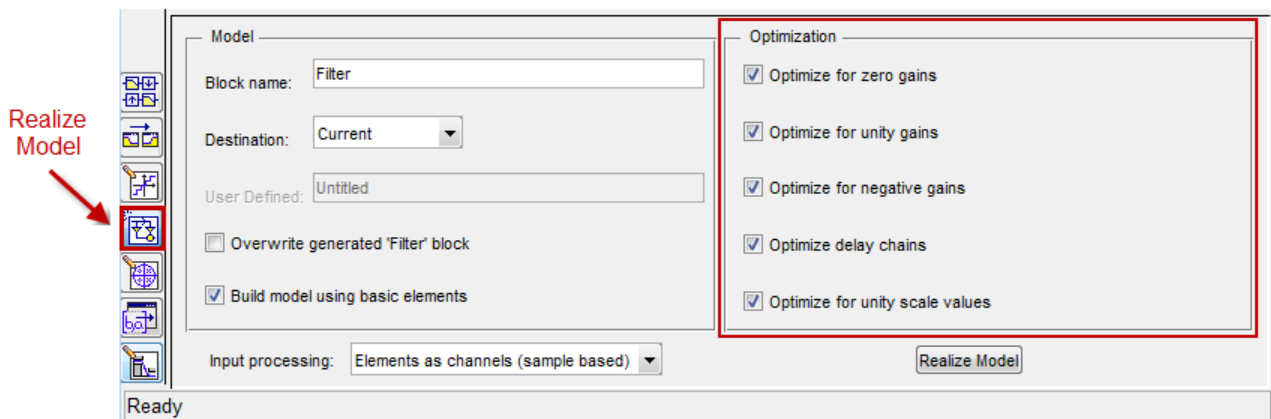
Optimize the Filter Structure

The Filter Realization Wizard can implement a digital filter using a Digital Filter block 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.

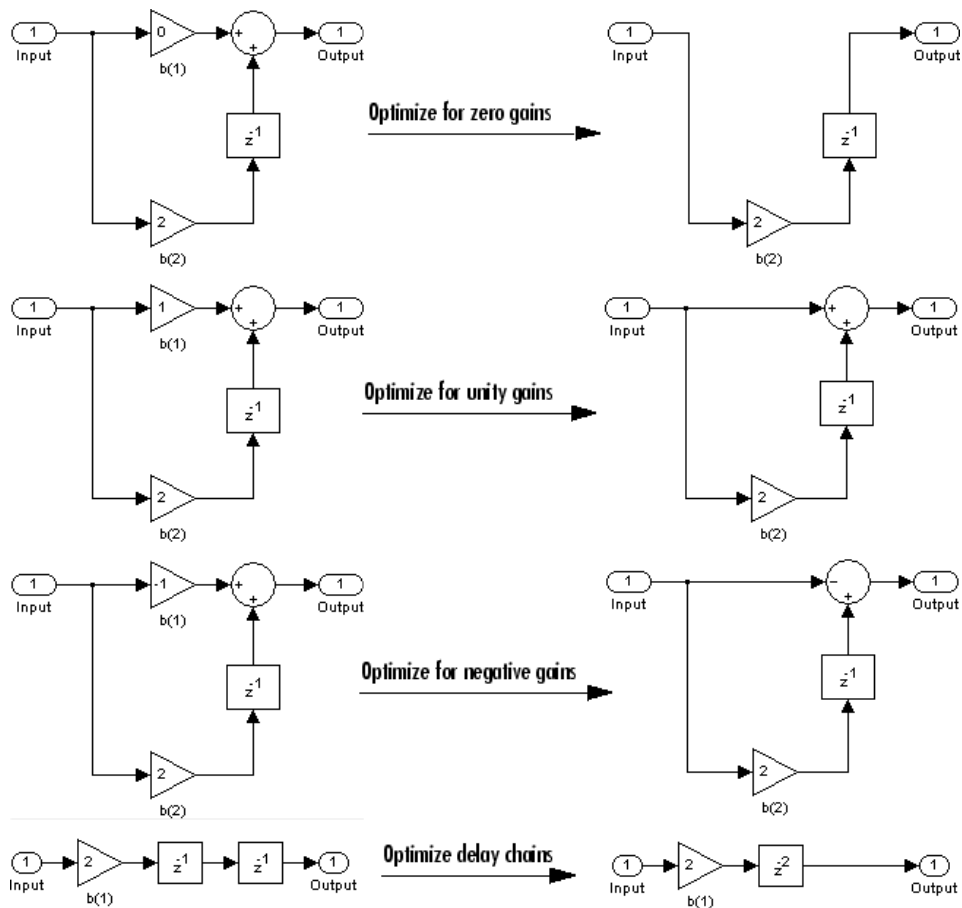
- 2 Select the desired optimizations in the **Optimization** region of the **Realize Model** pane. See the following descriptions and illustrations of each optimization option.



- **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 Block

In this section...

- “Overview of the Digital Filter Block” on page 3-147
- “Implement a Lowpass Filter in Simulink” on page 3-148
- “Implement a Highpass Filter in Simulink” on page 3-149
- “Filter High-Frequency Noise in Simulink” on page 3-150
- “Specify Static Filters” on page 3-155
- “Specify Time-Varying Filters” on page 3-155
- “Specify the SOS Matrix (Biquadratic Filter Coefficients)” on page 3-157

Overview of the Digital Filter Block

You can use the Digital Filter block to implement digital FIR and IIR filters in your models. Use this block if you have already performed the design and analysis and know your desired filter coefficients. You can use this block 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 filter block.

To implement a filter with the Digital Filter block, you must provide the following basic information about the filter:

- Whether the filter transfer function is FIR with all zeros, IIR with all poles, or IIR with poles and zeros
- The desired filter structure
- The filter coefficients

Note Use the Digital Filter Design block to design and implement a filter. Use the Digital Filter block to implement a pre-designed filter. Both blocks implement a filter in the same manner and have the same behavior during simulation and code generation.

Implement a Lowpass Filter in Simulink

You can use the Digital Filter block to implement a digital FIR or IIR filter. In this topic, you use it to implement an FIR lowpass filter:

- 1 Define the lowpass filter coefficients in the MATLAB workspace by typing

```
lopasNum = [-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 Digital Filter block into your model.
- 4 Double-click the Digital Filter block. Set the block parameters as follows, and then click **OK**:
 - **Coefficient source** = Dialog parameters
 - **Transfer function type** = FIR (all zeros)
 - **Filter structure** = Direct form transposed
 - **Numerator coefficients** = lopasNum
 - **Input processing** = Columns as channels (frame based)
 - **Initial conditions** = 0

Note that you can provide the filter coefficients in several ways:

- Type in a variable name from the MATLAB workspace, such as lopasNum.
- 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 Digital Filter block in your model now represents a lowpass filter. In the next topic, “Implement a Highpass Filter in Simulink” on page 3-149, you use a Digital Filter block to implement a highpass filter. For more information

about the Digital Filter block, see the Digital Filter block reference page. For more information about designing and implementing a new filter, see “Digital Filter Design Block” on page 3-122.

Implement a Highpass Filter in Simulink

In this topic, you implement an FIR highpass filter using the Digital Filter block:

- 1 If the model you created in “Implement a Lowpass Filter in Simulink” on page 3-148 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 Digital Filter block into your model.
- 4 Double-click the Digital Filter block. Set the block parameters as follows, and then click **OK**:
 - **Coefficient source** = Dialog parameters
 - **Transfer function type** = FIR (all zeros)
 - **Filter structure** = Direct form transposed
 - **Numerator coefficients** = hipassNum
 - **Input processing** = Columns as channels (frame based)
 - **Initial conditions** = 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 3-150, you use these Digital 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 3-122.

Filter High-Frequency Noise in Simulink

In the previous topics, you used Digital Filter blocks to implement 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 “Implement a Highpass Filter in Simulink” on page 3-149 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

```
lowpassNum = [-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];
```

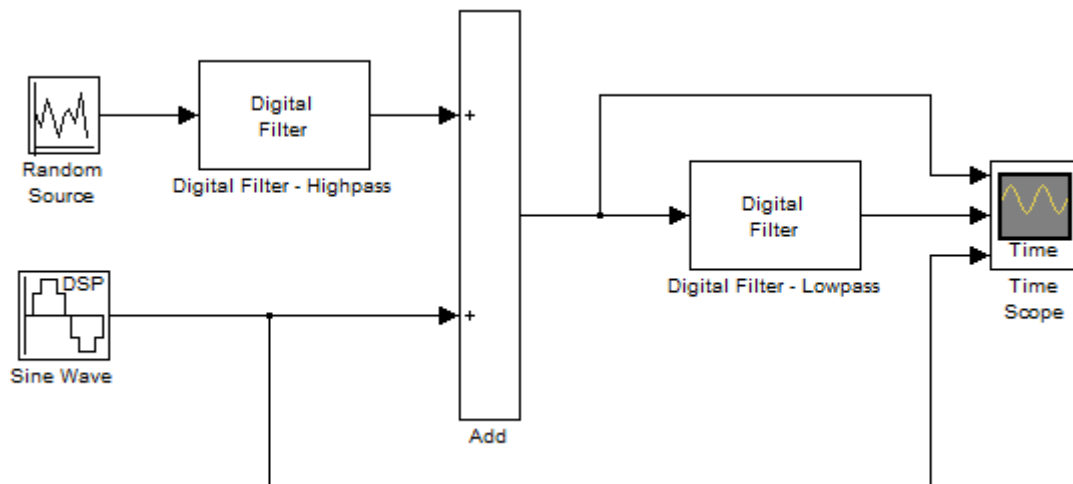

3 Click-and-drag the following blocks into your model file.

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	<ul style="list-style-type: none"> • Icon shape = rectangular • List of signs = ++
Random Source	<ul style="list-style-type: none"> • Source type = Uniform • Minimum = 0 • Maximum = 4 • Sample mode = Discrete • Sample time = 1/1000 • Samples per frame = 50
Sine Wave	<ul style="list-style-type: none"> • Frequency (Hz) = 75 • Sample time = 1/1000 • Samples per frame = 50
Time Scope	<ul style="list-style-type: none"> • File > Number of Input Ports > 3 • File > Configuration ... <ul style="list-style-type: none"> - 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 **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 **Start**.

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

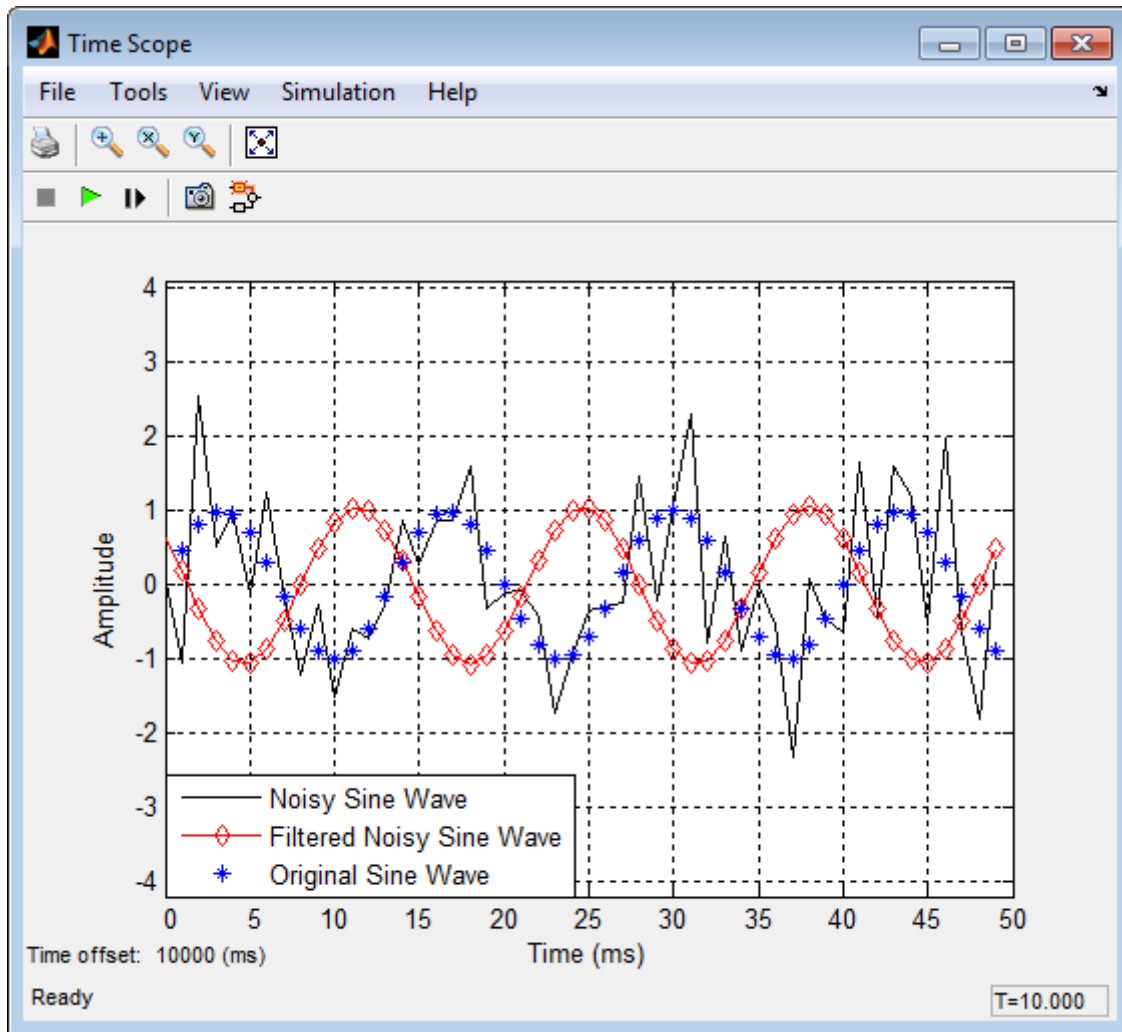
In the next step, you will set the color, style, and marker of each channel.

- 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 Digital 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 3-122.

Specify Static Filters

You can use the Digital Filter block to specify a static filter by setting the **Coefficient source** parameter to **Specify via dialog**. Depending on the filter structure, you need to enter your filter coefficients into one or more of the following parameters. The block disables all the irrelevant parameters. To see which of these parameters correspond to each filter structure, see “Supported Filter Structures” in *DSP System Toolbox Reference*:

- **Numerator coefficients** — Column or row vector of numerator coefficients, [b0, b1, b2, ..., bn].
- **Denominator coefficients** — Column or row vector of denominator coefficients, [a0, a1, a2, ..., am].
- **Reflection coefficients** — Column or row vector of reflection coefficients, [k1, k2, ..., kn].
- **SOS matrix (Mx6)** — M-by-6 SOS matrix. You can also use the Biquad Filter block to create a static biquadratic IIR filter.
- **Scale values** — Scalar or vector of M+1 scale values to be 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 vector(s) of filter coefficients, and click **OK**. You cannot change the filter order, so you cannot change the number of elements in the vector(s) of filter coefficients.

Specify Time-Varying Filters

Note This block does not support time-varying Biquadratic (SOS) filters.

Time-varying filters are filters whose coefficients change with time. You can specify a time-varying filter that changes once per frame or once per sample and you can filter multiple channels with each filter. However, you cannot apply different filters to each channel; all channels must be filtered with the same filter.

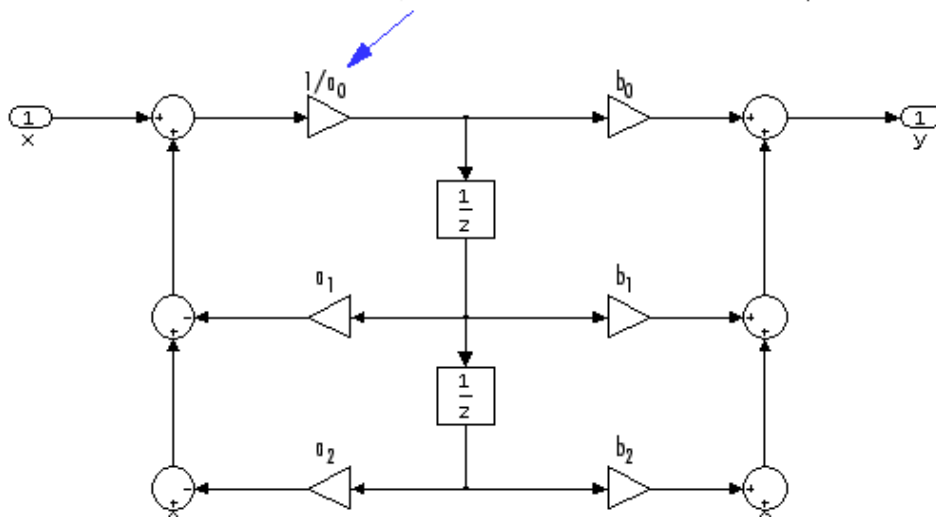
To specify a time-varying filter:

- 1** Set the **Coefficient source** parameter to `Input port(s)`, which enables extra block input ports for the time-varying filter coefficients.
- 2** Set the **Coefficient update rate** parameter to `One filter per frame` or `One filter per sample` depending on how often you want to update the filter coefficients.
- 3** Provide vectors of numerator, denominator, or reflection coefficients to the block input ports for filter coefficients. The series of vectors *must arrive at their ports at a specific rate, and must be of certain lengths*.
- 4** Select or clear the **First denominator coefficient = 1, remove a0 term in the structure** parameter depending on whether your first denominator coefficient is always 1. To learn more, see “Removing the a0 Term in the Filter Structure” on page 3-156.

Removing the a0 Term in the Filter Structure

When you know that the first denominator filter coefficient (a_0) is always 1 for your time-varying filter, select the **First denominator coefficient = 1, remove a0 term in the structure** parameter. Selecting this parameter reduces the number of computations the block must make to produce the output (the block omits the $1 / a_0$ term in the filter structure, as illustrated in the following figure). The block output is *invalid* if you select this parameter when the first denominator filter coefficient is *not always* 1 for your time-varying filter. Note that the block ignores the **First denominator coefficient = 1, remove a0 term in the structure** parameter for fixed-point inputs, since this block does not support nonunity a_0 coefficients for fixed-point inputs.

The block omits this term in the structure when you set the **First denominator coefficient = 1**, remove **a0** term in the structure parameter.



Specify the SOS Matrix (Biquadratic Filter Coefficients)

The Digital Filter block does not support *time-varying* biquadratic filters. To specify a *static* biquadratic filter (also known as a second-order section or SOS filter) using the Digital Filter Block, you need to set the following parameters as indicated:

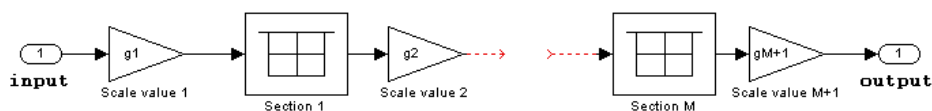
- **Transfer function type** — IIR (poles & zeros)
- **Filter structure** — Biquad direct form I (SOS), or Biquad direct form I transposed (SOS), or, or Biquad direct form II transposed (SOS)
- **SOS matrix (Mx6)** M-by-6 SOS matrix

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.

Note You can also use the Biquad Filter block to implement a static biquadratic IIR filter.

Analog Filter Design Block

The Analog Filter Design block designs and implements analog IIR filters with standard band configurations. All of the analog filter designs let you specify a filter order. The other available parameters depend on the filter type and band configuration, as shown in the following table.

Configuration	Butterworth	Chebyshev I	Chebyshev II	Elliptic
Lowpass	Ω_p	Ω_p, R_p	Ω_s, R_s	Ω_p, R_p, R_s
Highpass	Ω_p	Ω_p, R_p	Ω_s, R_s	Ω_p, R_p, R_s
Bandpass	Ω_{p1}, Ω_{p2}	$\Omega_{p1}, \Omega_{p2}, R_p$	$\Omega_{s1}, \Omega_{s2}, R_s$	$\Omega_{p1}, \Omega_{p2}, R_p, R_s$
Bandstop	Ω_{p1}, Ω_{p2}	$\Omega_{p1}, \Omega_{p2}, R_p$	$\Omega_{s1}, \Omega_{s2}, R_s$	$\Omega_{p1}, \Omega_{p2}, R_p, R_s$

The table parameters are

- Ω_p — passband edge frequency
- Ω_{p1} — lower passband edge frequency
- Ω_{p2} — upper cutoff frequency
- Ω_s — stopband edge frequency
- Ω_{s1} — lower stopband edge frequency
- Ω_{s2} — upper stopband edge frequency
- R_p — passband ripple in decibels
- R_s — stopband attenuation in decibels

For all of the analog filter designs, frequency parameters are in units of radians per second.

The Analog Filter Design block uses a state-space filter representation, and applies the filter using the State-Space block in the Simulink Continuous library. All of the design methods use Signal Processing Toolbox functions to design the filter:

- The Butterworth design uses the toolbox function `butter`.
- The Chebyshev type I design uses the toolbox function `cheby1`.
- The Chebyshev type II design uses the toolbox function `cheby2`.
- The elliptic design uses the toolbox function `ellip`.

The Analog Filter Design block is built on the filter design capabilities of Signal Processing Toolbox software. For more information on the filter design algorithms, see “Filter Design and Implementation” in the Signal Processing Toolbox documentation.

Note The Analog Filter Design block does not work with the Simulink discrete solver, which is enabled when the **Solver** list is set to `Discrete` (no continuous states) in the **Solver** pane of the **Configuration Parameters** dialog box. Select one of the continuous solvers (such as `ode4`) instead.

Fixed-Point Filter Design

In this section...

“Overview of Fixed-Point Filters” on page 8-65

“Data Types for Filter Functions” on page 8-65

“Convert a Filter from Floating Point to Fixed Point in MATLAB” on page 8-67

“Create an FIR Filter Using Integer Coefficients” on page 8-75

“Fixed-Point Filtering in Simulink” on page 8-92

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 math, see “Fixed-Point Concepts” in “Fixed-Point Toolbox” documentation.

Data Types for Filter Functions

- “Data Type Support” on page 8-65
- “Fixed Data Type Support” on page 8-66
- “Single Data Type Support” on page 8-66

Data Type Support

There are three different data types supported in DSP System Toolbox software:

- Fixed — Requires Fixed Point Toolbox and is supported by packages listed in “Fixed Data Type Support” on page 8-66.
- 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 8-66.

Fixed Data Type Support

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

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

To set the filter’s arithmetic property:

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

Single Data Type Support

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

Convert a Filter from Floating Point to Fixed Point in MATLAB

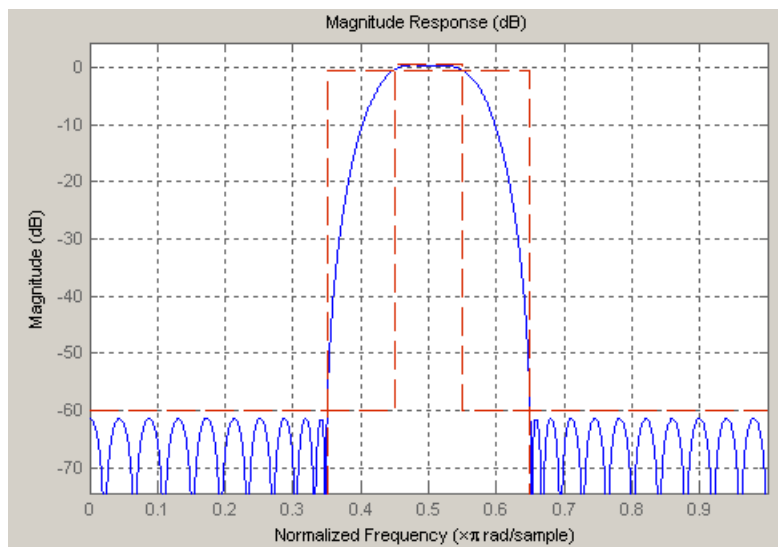
- “Process Overview” on page 8-67
- “Design the Filter” on page 8-67
- “Quantize the Coefficients” on page 8-68
- “Dynamic Range Analysis” on page 8-71
- “Compare Magnitude Response and Magnitude Response Estimate” on page 8-72

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.

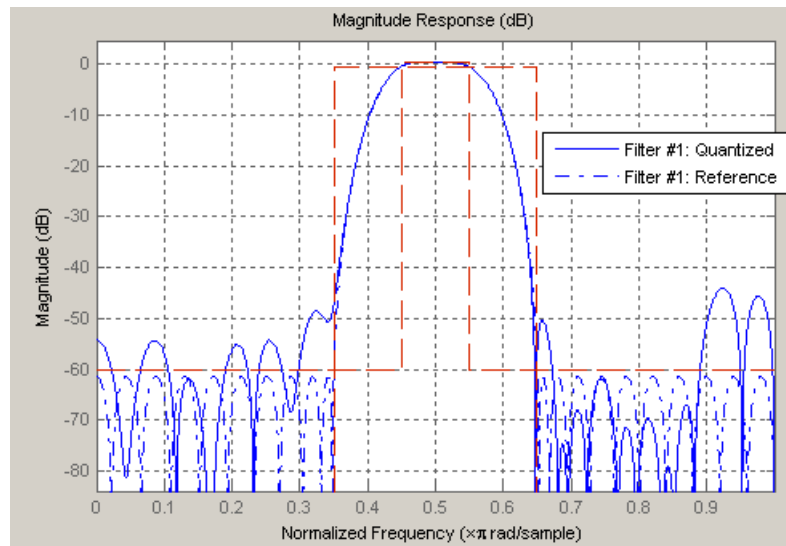
Quantize the Coefficients

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.

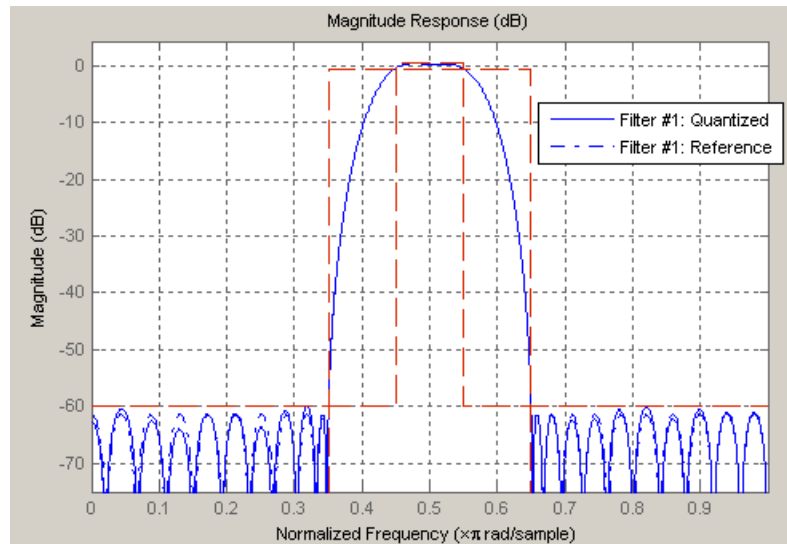


As the figure shows, the filter design constraints are not met. The attenuation is not complete, and there is noise at the edges of the stopbands. You can experiment with different coefficient word lengths if you like. For this example, however, the word length of 12 is sufficient.

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');
```

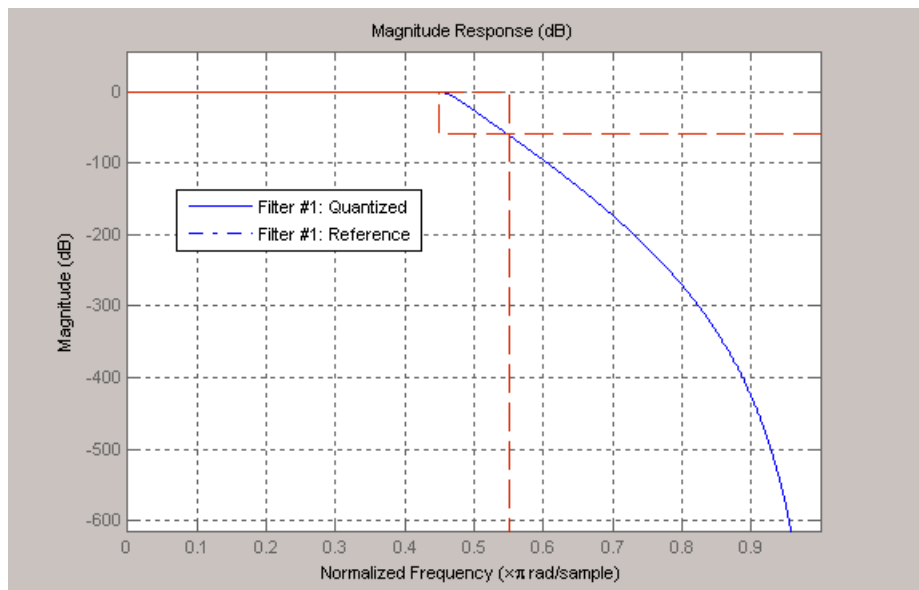
Now set the arithmetic to fixed-point.

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

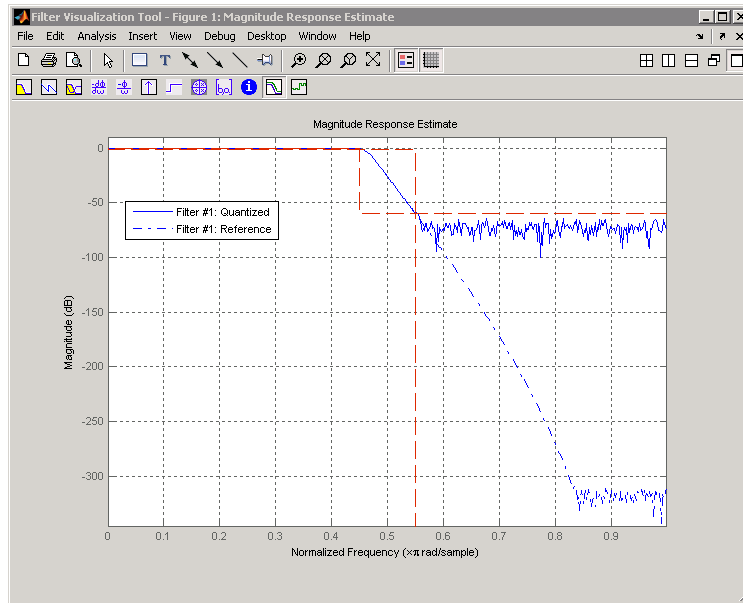
Open the filter using fvtool.

```
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 F_s . For more information about analyzing filters in this way, refer to the section titled Analyzing Filters with a Noise-Based Method in the User Guide.

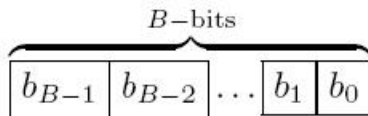
For more information, refer to McClellan, et al., Computer-Based Exercises for Signal Processing Using MATLAB 5, Prentice-Hall, 1998. See Project 5: Quantization Noise in Digital Filters, page 231.

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:

$$\boxed{b_{B-1}} \boxed{b_{B-2}} \dots \boxed{b_1} \boxed{b_0}$$

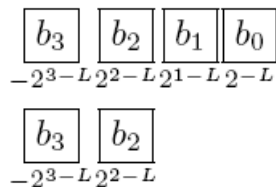
$$-2^{B-1-L} \quad 2^{B-2-L} \quad \dots \quad 2^{1-L} \quad 2^{-L}$$

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

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

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

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



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

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

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

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 $\{-32768, -32512, \dots, -256, 0, 256, \dots, 32256, 32512\}$.

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

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

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

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

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

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

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

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

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:

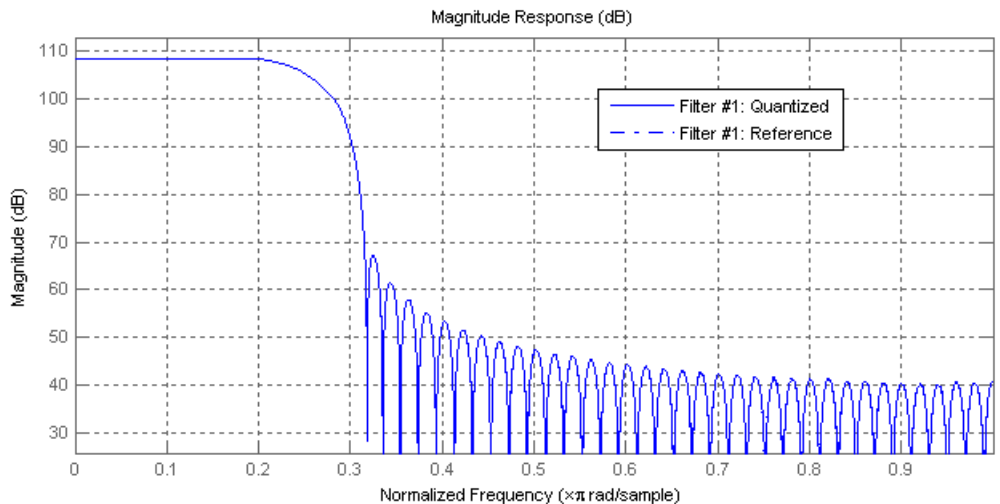
```
all(h.Numerator == round(h.Numerator))
```

```
ans =
```

```
1
```

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

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



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

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

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:

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

Here `y_sc` is a full precision output, meaning that no bits have been discarded in the computation. This makes `y_sc` 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:

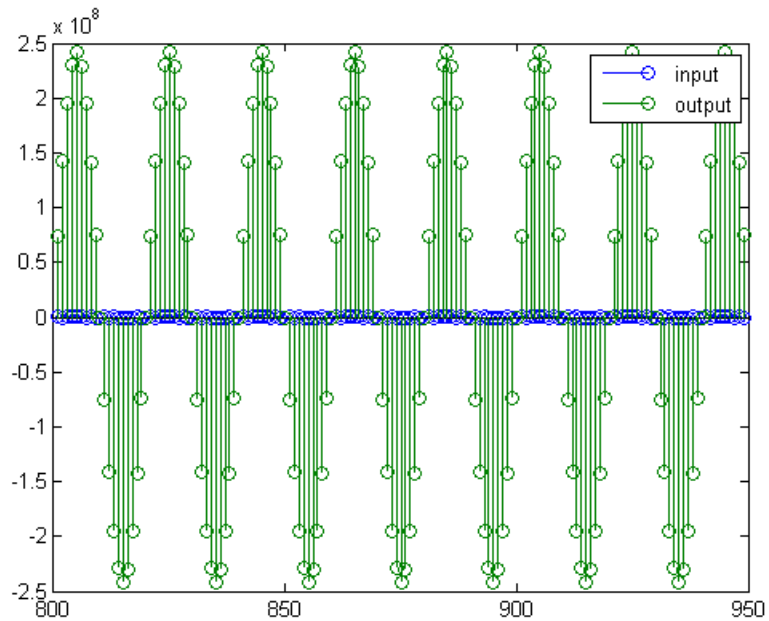
```
hd = double(h);
xd = double(xsc);
yd = filter(hd, xd);
norm(yd-double(y_sc))
```

```
ans =
```

```
0
```

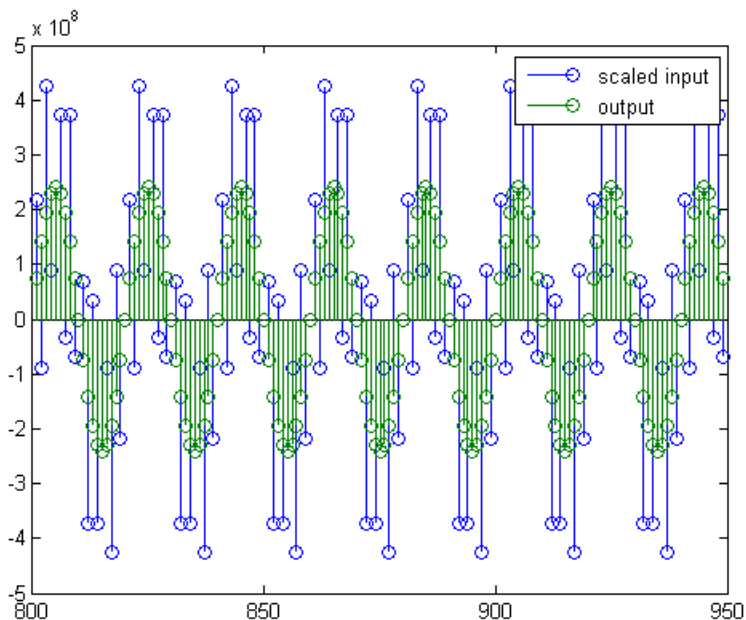
Now you can examine the output compared to the input. This example is plotting only the last few samples to minimize the effect of transients:

```
idx = 800:950;
xs_cext = double(xsc(idx));
gd = grpdelay(h, [f1 f2]);
yidx = idx + gd(1);
ys_cext = double(y_sc(yidx));
stem(n(idx)', [xs_cext, ys_cext]);
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*xscent, yscent]);
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 8-75 the 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:

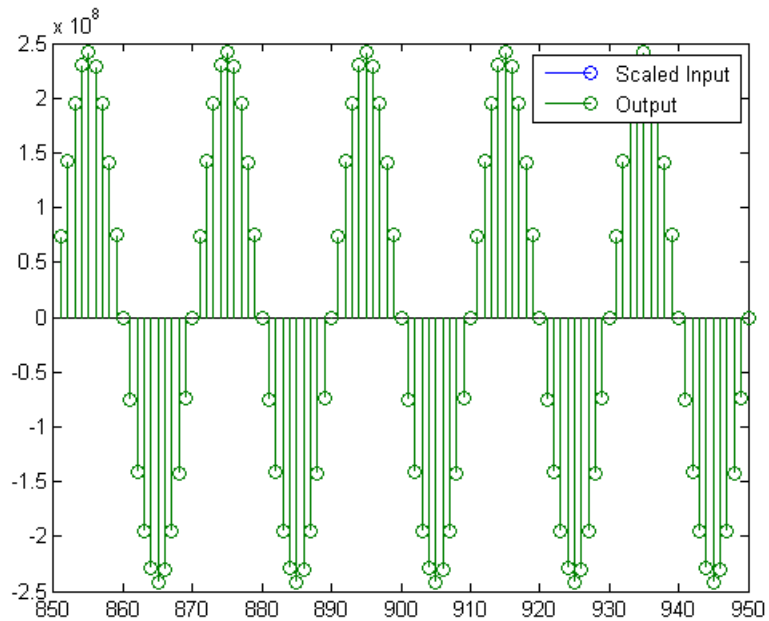
```
norm(double(yout) - double(ysc))
```

ans =

1.446323386867543e+005

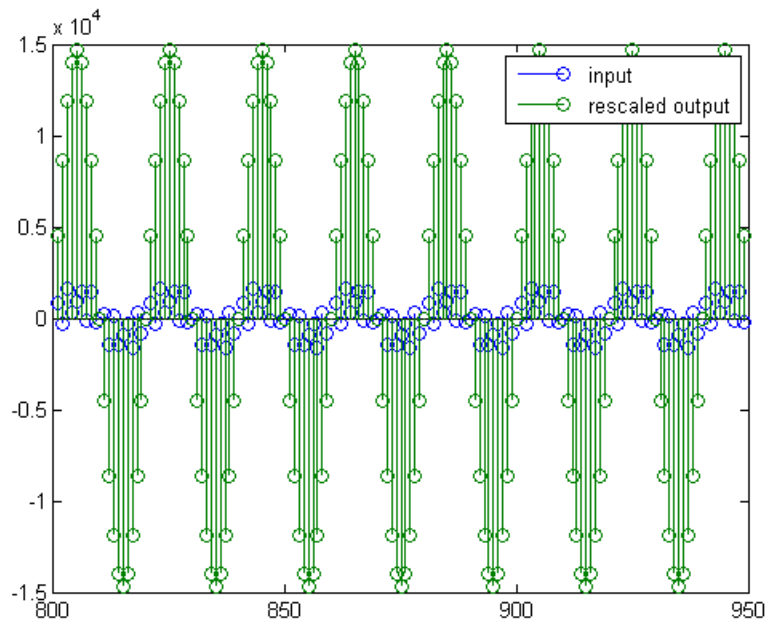
In this case the differences are hard to spot when plotting the data, as seen below:

```
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:

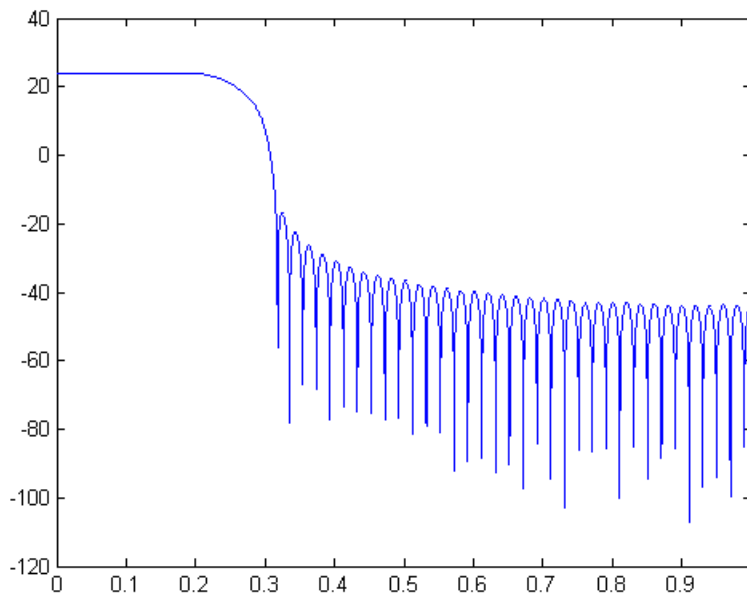
```
yri = fi(yout.int, true, 16, 0);
stem(n(idx)', [xsceft, double(yri(yidx)')]);
axis([800 950 -1.5e4 1.5e4]);
legend('input', 'rescaled output');
```



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:

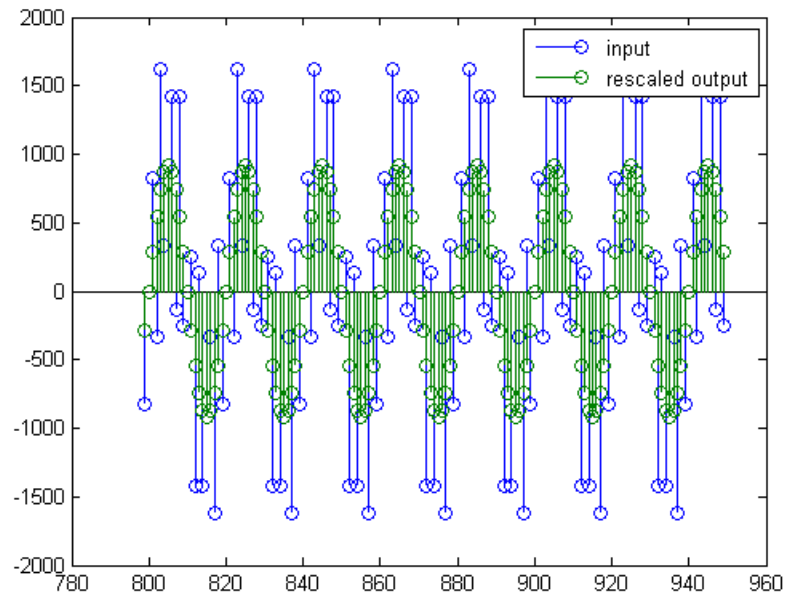
```
[H,w] = freqz(h);  
plot(w/pi, 20*log10(2^(-14)*abs(H)));
```



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

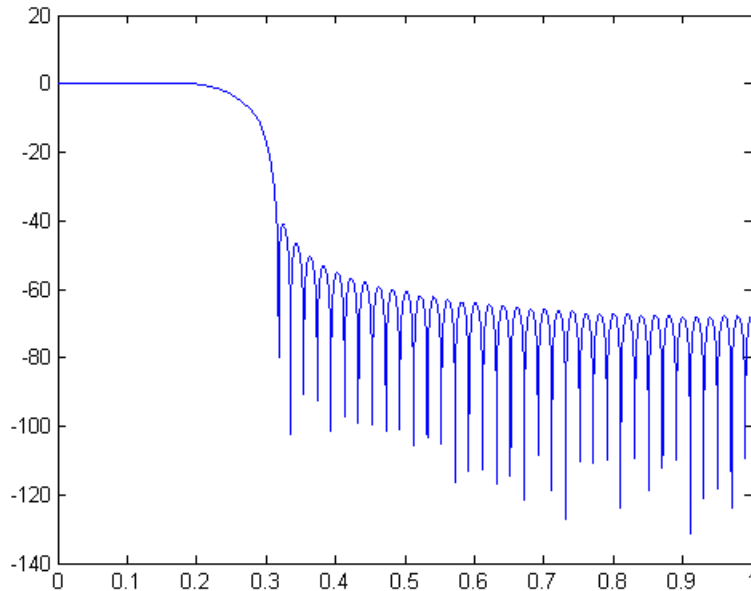
To put the input and output on the same scale, the MSBs must be weighted equally. The input MSB has a weight of 2^{11} , whereas the scaled output MSB has a weight of $2^{(29-14)} = 2^{15}$. You need to give the output MSB a weight of 2^{11} as follows:

```
yf = fi(zeros(size(yri)), true, 16, 4);  
yf.bin = yri.bin;  
stem(n(idx)', [xscent, 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” on page 8-89
- “Reinterpret the Output” on page 8-90

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.

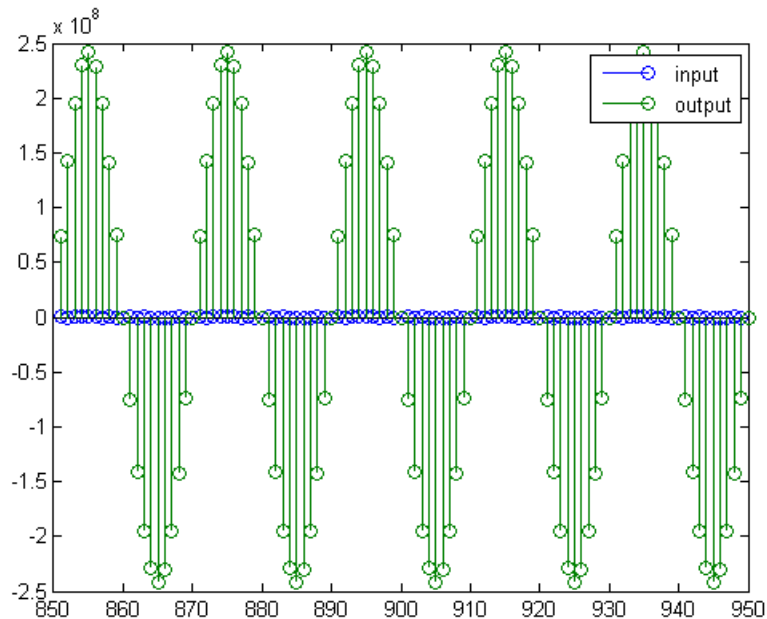
```
yint = filter(h, xsc);
norm(yd - double(yint))
```

```
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');
```



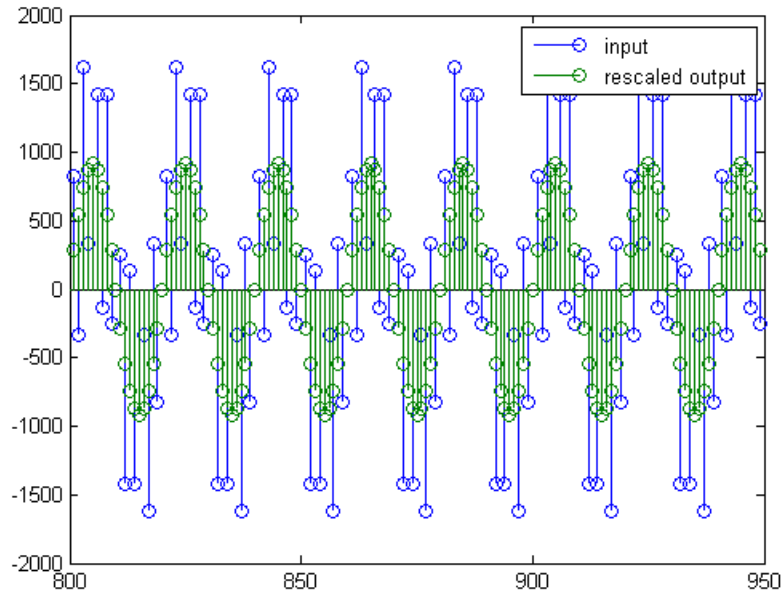
Once again, the plot shows that the input and output are at different scales. In order to scale the output so that the signals can be compared more easily in a plot, you will need to weigh the MSBs appropriately. You can compute the new fraction length using the gain of the filter when the coefficients 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)', [xscent, 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 8-92
- “Filter Implementation Blocks” on page 8-93
- “Filter Design and Implementation Blocks” on page 8-93

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

- Digital Filter
- 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, Two-Channel Synthesis Subband Filter, and Digital Filter blocks are all implementation blocks. They allow you to implement filters for which you already know the filter coefficients. The first four blocks each implement their respective filter type, while the Digital Filter block can create a variety of filter structures. All filter structures supported by the Digital Filter block support fixed-point signals.

For more information on these filter implementation blocks, see their reference pages in the Block Reference.

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.

Adaptive Filters

Learn how to design and implement adaptive filters.

- “Overview of Adaptive Filters and Applications” on page 4-2
- “Adaptive Filters in DSP System Toolbox Software” on page 4-10
- “LMS Adaptive Filters” on page 4-14
- “RLS Adaptive Filters” on page 4-36
- “Enhance a Signal Using LMS and Normalized LMS Algorithms” on page 4-42
- “Adaptive Filters in Simulink” on page 4-51
- “Selected Bibliography” on page 4-64

Overview of Adaptive Filters and Applications

In this section...

“Introduction to Adaptive Filtering” on page 4-2

“Adaptive Filtering Methodology” on page 4-2

“Choosing an Adaptive Filter” on page 4-4

“System Identification” on page 4-6

“Inverse System Identification” on page 4-6

“Noise or Interference Cancellation” on page 4-7

“Prediction” on page 4-8

Introduction to Adaptive Filtering

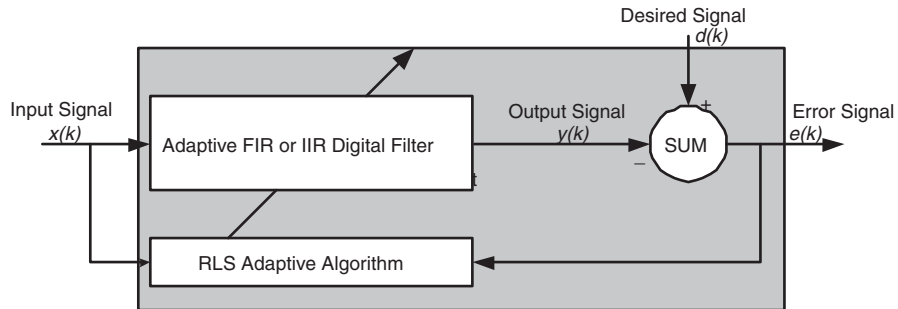
Adaptive filtering involves the changing of filter parameters (coefficients) over time, to adapt to changing signal characteristics. Over the past three decades, digital signal processors have made great advances in increasing speed and complexity, and reducing power consumption. As a result, real-time 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 4-64.

Adaptive Filtering Methodology

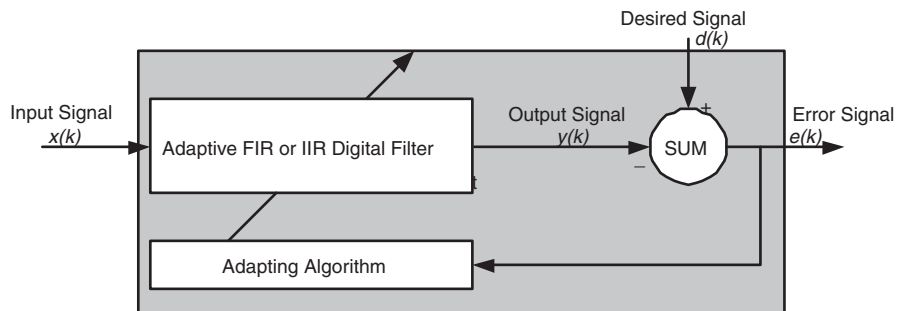
This section presents a brief description of how adaptive filters work and some of the applications where they can be useful.

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 `adaptfilt.lms`, `adaptfilt.filtx1ms`, and `adaptfilt.nlms`
- Adaptive filters based on recursive least squares (RLS) techniques. For example, `adaptfilt.rls` and `adaptfilt.swrls`
- Adaptive filters based on sign-data (`adaptfilt.sd`), sign-error (`adaptfilt.se`), and sign-sign (`adaptfilt.ss`) techniques

- Adaptive filters based on lattice filters. For example, `adaptfilt.gal` and `adaptfilt.lsl`
- Adaptive filters that operate in the frequency domain, such as `adaptfilt.fdaf` and `adaptfilt.pbufdaf`.
- Adaptive filters that operate in the transform domain. Two of these are the `adaptfilt.tdafdft` and `adaptfilt.tdafdct` filters

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

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

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

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

“Adaptive Filters in DSP System Toolbox Software” on page 4-10 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.

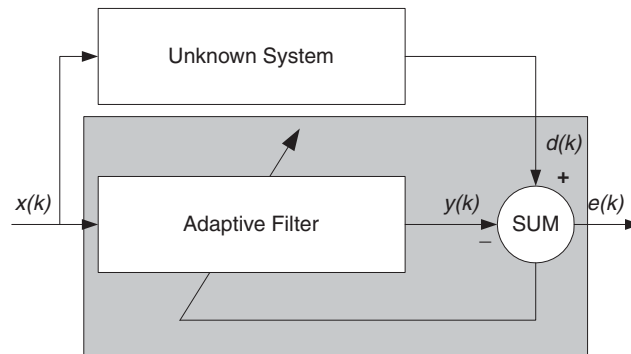
Additionally, starting with an LMS approach can form a solid baseline against which you can study and compare the more complex adaptive filters available in the toolbox. Finally, your development process should, at some time, test

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.



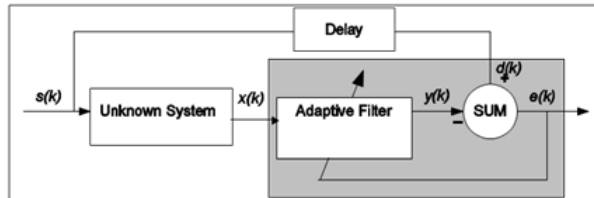
Using an Adaptive Filter to Identify an Unknown System

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

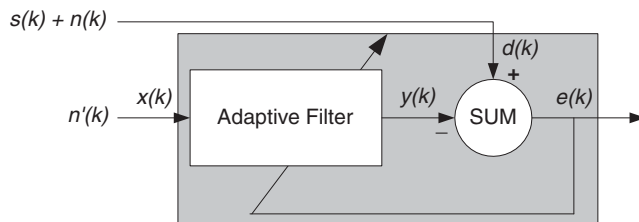
Including the delay to account for the delay caused by the unknown system prevents this condition.

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.

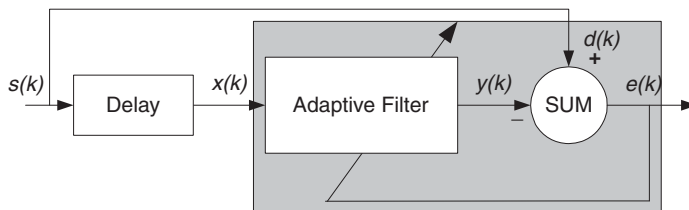


Using an Adaptive Filter to Remove Noise from an Unknown System

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 4-10
“Algorithms” on page 4-10
“Using Adaptive Filter Objects” on page 4-13

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

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

Least Mean Squares (LMS) Based FIR Adaptive Filters

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

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

Recursive Least Squares (RLS) Based FIR Adaptive Filters

Adaptive Filter Method	Adapting Algorithm Used to Generate Filter Coefficients During Adaptation
<code>adaptfilt.ftf</code>	Fast transversal least-squares adaptation algorithm
<code>adaptfilt.qrdrls</code>	QR-decomposition RLS adaptation algorithm
<code>adaptfilt.hrls</code>	Householder RLS adaptation algorithm
<code>adaptfilt.hswrls</code>	Householder SWRLS adaptation algorithm
<code>adaptfilt.rls</code>	Recursive-least squares (RLS) adaptation algorithm
<code>adaptfilt.swrls</code>	Sliding window (SW) RLS adaptation algorithm
<code>adaptfilt.swftf</code>	Sliding window FTF adaptation algorithm

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

Affine Projection (AP) FIR Adaptive Filters

Adaptive Filter Method	Adapting Algorithm Used to Generate Filter Coefficients During Adaptation
adaptfilt.ap	Affine projection algorithm that uses direct matrix inversion
adaptfilt.apru	Affine projection algorithm that uses recursive matrix updating
adaptfilt.bap	Block affine projection adaptation algorithm

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

FIR Adaptive Filters in the Frequency Domain (FD)

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

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

Lattice-Based (L) FIR Adaptive Filters

Adaptive Filter Method	Description of the Adapting Algorithm Used to Generate Filter Coefficients During Adaptation
<code>adaptfilt.gal</code>	Gradient adaptive lattice filter adaptation algorithm
<code>adaptfilt.lsl</code>	Least squares lattice adaptation algorithm
<code>adaptfilt.qrdsl</code>	QR decomposition RLS adaptation algorithm

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

Presenting a detailed derivation of the Wiener-Hopf equation and determining solutions to it is beyond the scope of this *User's Guide*. Full descriptions of the theory appear in the adaptive filter references provided in the “Selected Bibliography” on page 4-64.

Using Adaptive Filter Objects

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

- “LMS Adaptive Filters” on page 4-14
- “RLS Adaptive Filters” on page 4-36

LMS Adaptive Filters

In this section...

“LMS Methods for `adaptfilt` Objects” on page 4-14

“System Identification Using `adaptfilt.lms`” on page 4-16

“System Identification Using `adaptfilt.nlms`” on page 4-19

“Noise Cancellation Using `adaptfilt.sd`” on page 4-22

“Noise Cancellation Using `adaptfilt.se`” on page 4-26

“Noise Cancellation Using `adaptfilt.ss`” on page 4-31

LMS Methods for `adaptfilt` Objects

This section provides introductory examples using some of the least mean squares (LMS) adaptive filter functions in the toolbox.

The toolbox provides many adaptive filter design functions that use the LMS algorithms to search for the optimal solution to the adaptive filter, including

- `adaptfilt.lms` — Implement the LMS algorithm to solve the Wiener-Hopf equation and find the filter coefficients for an adaptive filter.
- `adaptfilt.nlms` — Implement the normalized variation of the LMS algorithm to solve the Wiener-Hopf equation and determine the filter coefficients of an adaptive filter.
- `adaptfilt.sd` — Implement the sign-data variation of the LMS algorithm to solve the Wiener-Hopf equation and determine the filter coefficients of an adaptive filter. The correction to the filter weights at each iteration depends on the sign of the input $x(k)$.
- `adaptfilt.se` — Implement the sign-error variation of the LMS algorithm to solve the Wiener-Hopf equation and determine the filter coefficients of an adaptive filter. The correction applied to the current filter weights for each successive iteration depends on the sign of the error, $e(k)$.
- `adaptfilt.ss` — Implement the sign-sign variation of the LMS algorithm to solve the Wiener-Hopf equation and determine the filter coefficients of an

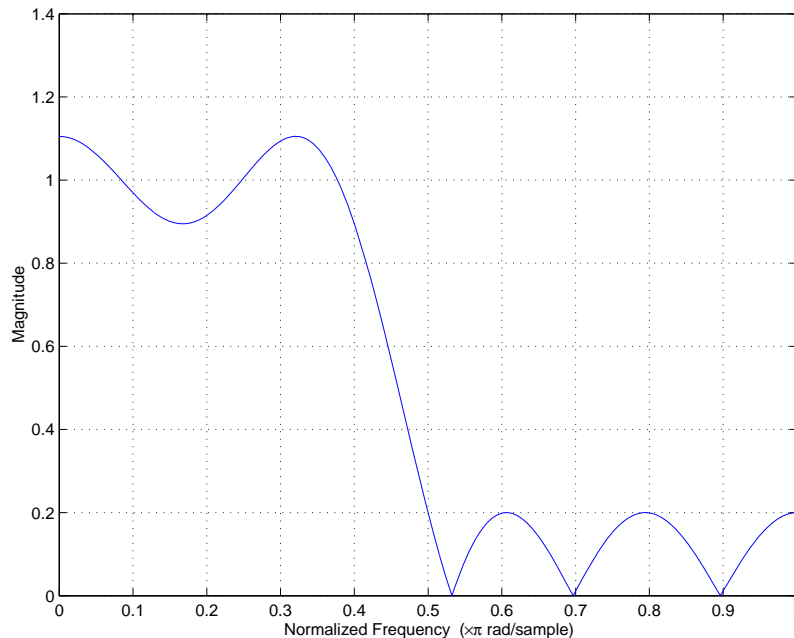
adaptive filter. The correction applied to the current filter weights for each successive iteration depends on both the sign of $x(k)$ and the sign of $e(k)$.

To demonstrate the differences and similarities among the various LMS algorithms supplied in the toolbox, the LMS and NLMS adaptive filter examples use the same filter for the unknown system. The unknown filter is the constrained lowpass filter from `firgr` and `firband` examples.

```
[b,err,res]=firgr(12,[0 0.4 0.5 1], [1 1 0 0], [1 0.2],...
{'w' 'c'});
```

From the figure you see that the filter is indeed lowpass and constrained to 0.2 ripple in the stopband. With this as the baseline, the adaptive LMS filter examples use the adaptive LMS algorithms and their initialization functions to identify this filter in a system identification role.

To review the general model for system ID mode, look at “System Identification” on page 4-6 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 `adaptfilt.lms`

To use the adaptive filter functions in the toolbox you need to provide three things:

- The adaptive LMS function to use. This example uses the LMS adaptive filter function `adaptfilt.lms`.
- An unknown system or process to adapt to. In this example, the filter designed by `firgr` is the unknown system.
- Appropriate input data to exercise the adaptation process. In terms of the generic LMS model, these are the desired signal $d(k)$ and the input signal $x(k)$.

Start by defining an input signal `x`.

```
x = 0.1*randn(1,250);
```

The input is broadband noise. For the unknown system filter, use `firgr` to create a twelfth-order lowpass filter:

```
[b,err,res] = firgr(12,[0 0.4 0.5 1],[1 1 0 0],[1 0.2],{'w','c'});
```

Although you do not need them here, include the `err` and `res` output arguments.

Now filter the signal through the unknown system to get the desired signal.

```
d = filter(b,1,x);
```

With the unknown filter designed and the desired signal in place you construct and apply the adaptive LMS filter object to identify the unknown.

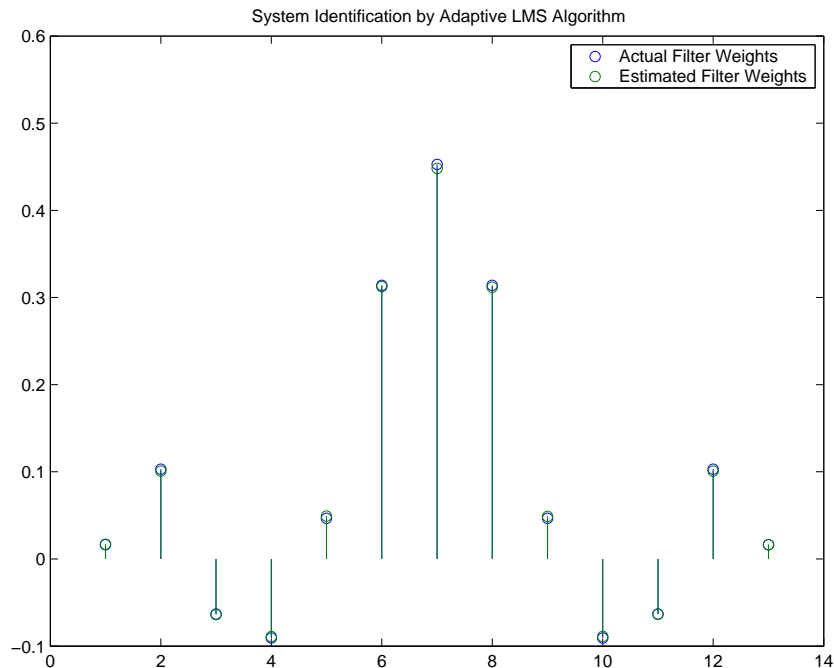
Preparing the adaptive filter object requires that you provide starting values for estimates of the filter coefficients and the LMS step size. You could start with estimated coefficients of some set of nonzero values; this example uses zeros for the 12 initial filter weights.

For the step size, 0.8 is a reasonable value — a good compromise between being large enough to converge well within the 250 iterations (250 input sample points) and small enough to create an accurate estimate of the unknown filter.

```
mu = 0.8;
ha = adaptfilt.lms(13,mu);
```

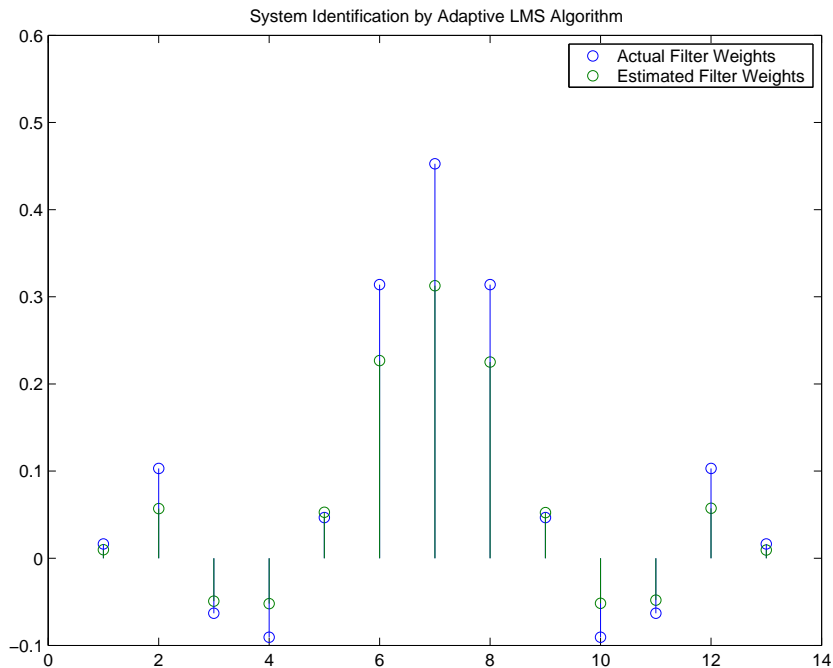
Finally, using the `adaptfilt` object `ha`, desired signal, `d`, and the input to the filter, `x`, run the adaptive filter to determine the unknown system and plot the results, comparing the actual coefficients from `firgr` to the coefficients found by `adaptfilt.lms`.

```
[y,e] = filter(ha,x,d);
stem([b.' ha.coefficients.'])
```

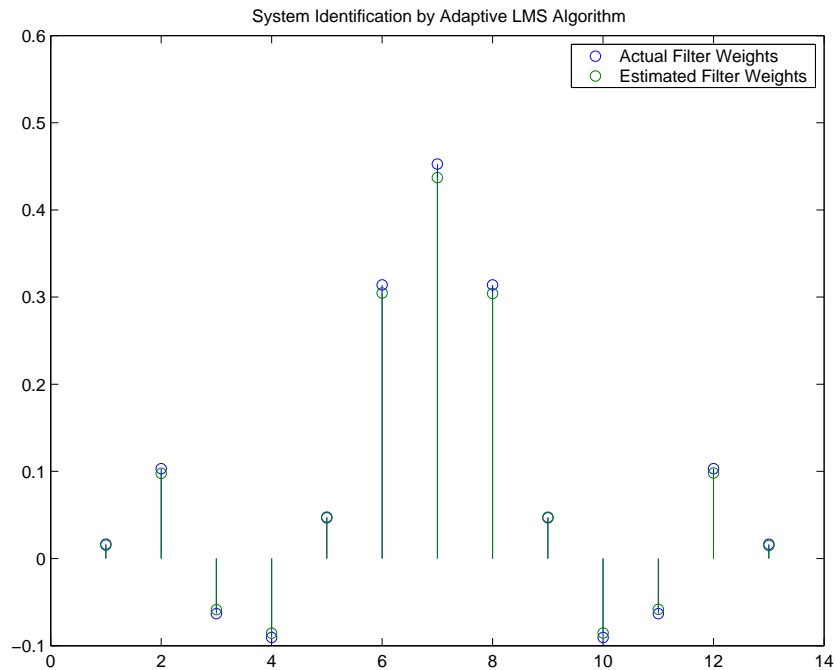


In the stem plot the actual and estimated filter weights are the same. As an experiment, try changing the step size to 0.2. Repeating the example with

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



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



System Identification Using `adaptfilt.nlm`s

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 `adaptfilt.nlm`s example, you used `firgr` to create the filter that you would identify. So you can compare the results, you use the same filter, and replace `adaptfilt.lms` with `adaptfilt.nlm`s, to use the normalized LMS algorithm variation. You should see better convergence with similar fidelity.

First, generate the input signal and the unknown filter.

```
x = 0.1*randn(1,500);  
[b,err,res] = fircband(12,[0 0.4 0.5 1], [1 1 0 0], [1 0.2],...  
{'w' 'c'});  
d = filter(b,1,x);
```

Again d represents the desired signal $d(x)$ as you defined it earlier and b contains the filter coefficients for your unknown filter.

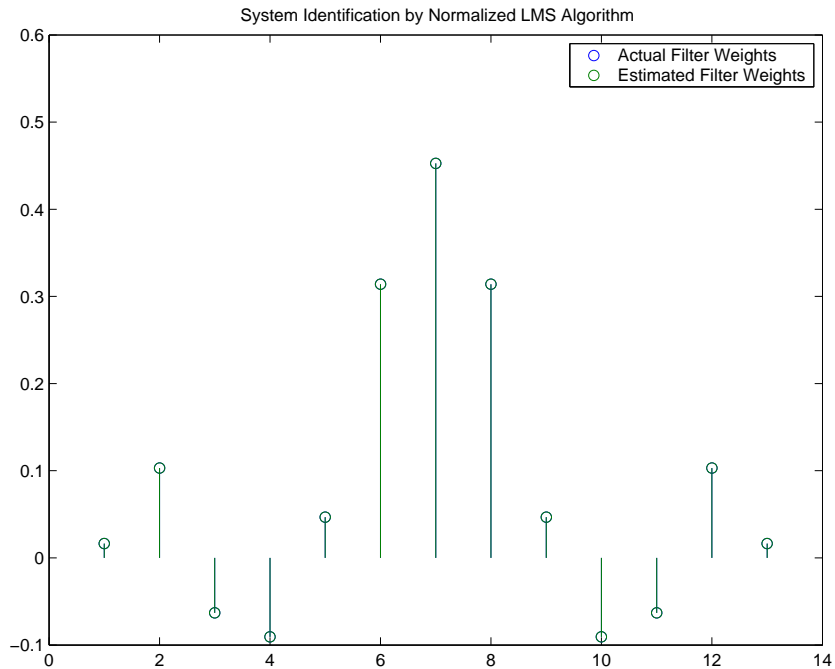
```
mu = 0.8;  
ha = adaptfilt.nlms(13,mu);
```

You use the preceding code to initialize the normalized LMS algorithm. For more information about the optional input arguments, refer to `adaptfilt.nlms` in the reference section of this *User's Guide*.

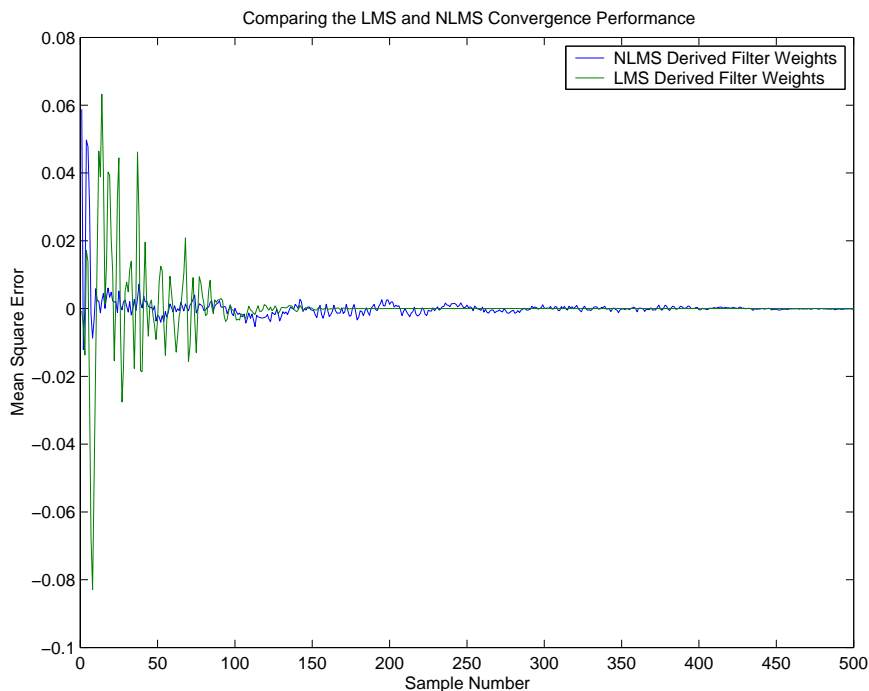
Running the system identification process is a matter of using `adaptfilt.nlms` with the desired signal, the input signal, and the initial filter coefficients and conditions specified in `s` as input arguments. Then plot the results to compare the adapted filter to the actual filter.

```
[y,e] = filter(ha,x,d);  
stem([b.' ha.coefficients.'])
```

As shown in the following stem plot (a convenient way to compare the estimated and actual filter coefficients), the two are nearly identical.



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



Noise Cancellation Using `adaptfilt.sd`

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

$$w(k+1) = w(k) + \mu e(k) \operatorname{sgn}[x(k)],$$

$$\operatorname{sgn}[x(k)] = \begin{cases} 1, & x(k) > 0 \\ 0, & x(k) = 0 \\ -1, & x(k) < 0 \end{cases}$$

with vector \mathbf{w} containing the weights applied to the filter coefficients and vector \mathbf{x} containing the input data. $e(k)$ (equal to desired signal - filtered signal) is the error at time k and is the quantity the SDLMS algorithm seeks to minimize. μ (μ) is the step size.

As you specify μ smaller, the correction to the filter weights gets smaller for each sample and the SDLMS error falls more slowly. Larger μ 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 μ within the following practical bounds

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

where N is the number of samples in the signal. Also, define μ as a power of two for efficient computing.

Note How you set the initial conditions of the sign-data algorithm profoundly influences the effectiveness of the adaptation. Because the algorithm essentially quantizes the input signal, the algorithm can become unstable easily.

A series of large input values, coupled with the quantization process may result in the error growing beyond all bounds. You restrain the tendency of the sign-data algorithm to get out of control by choosing a small step size ($\mu \ll 1$) and setting the initial conditions for the algorithm to nonzero positive and negative values.

In this noise cancellation example, `adaptfilt.sd` requires two input data sets:

- Data containing a signal corrupted by noise. In Using an Adaptive Filter to Remove Noise from an Unknown System on page 4-8, this is $d(k)$, the desired signal. The noise cancellation process removes the noise, leaving the signal.
- Data containing random noise ($x(k)$ in Using an Adaptive Filter to Remove Noise from an Unknown System on page 4-8) that is correlated with the noise that corrupts the signal data. Without the correlation between the noise data, the adapting algorithm cannot remove the noise from the signal.

For the signal, use a sine wave. Note that `signal` is a column vector of 1000 elements.

```
signal = sin(2*pi*0.055*[0:1000-1]');
```

Now, add correlated white noise to `signal`. To ensure that the noise is correlated, pass the noise through a lowpass FIR filter, and then add the filtered noise to the signal.

```
noise=randn(1,1000);  
nfilt=fir1(11,0.4); % Eleventh order lowpass filter  
fnoise=filter(nfilt,1,noise); % Correlated noise data  
d=signal.'+fnoise;
```


`fnoise` is the correlated noise and `d` is now the desired input to the sign-data algorithm.

To prepare the `adaptfilt` object for processing, set the input conditions `coeffs` and `mu` for the object. As noted earlier in this section, the values you set for `coeffs` and `mu` determine whether the adaptive filter can remove the noise from the signal path.

In “System Identification Using `adaptfilt.lms`” on page 4-16, you constructed a default filter that sets the filter coefficients to zeros. In most cases that approach does not work for the sign-data algorithm. The closer you set your initial filter coefficients to the expected values, the more likely it is that the algorithm remains well behaved and converges to a filter solution that removes the noise effectively.

For this example, start with the coefficients in the filter you used to filter the noise (`nfilt`), and modify them slightly so the algorithm has to adapt.

```
coeffs = nfilt.' -0.01; % Set the filter initial conditions.  
mu = 0.05;           % Set the step size for algorithm updating.
```

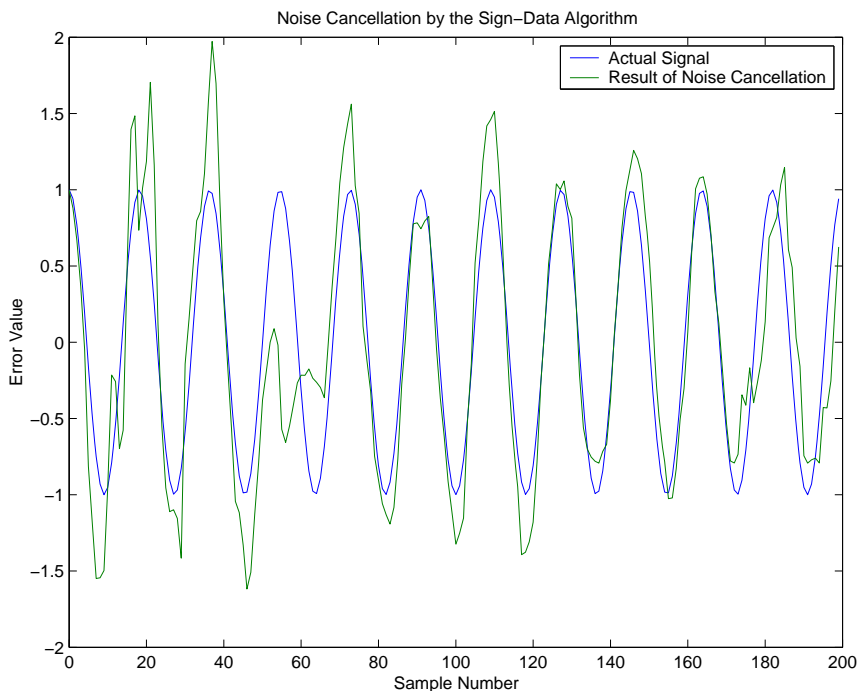
With the required input arguments for `adaptfilt.sd` prepared, construct the `adaptfilt` object, run the adaptation, and view the results.

```
ha = adaptfilt.sd(12,mu)  
set(ha,'coefficients',coeffs);  
[y,e] = filter(ha,noise,d);  
plot(0:199,signal(1:200),0:199,e(1:200));
```

When `adaptfilt.sd` runs, it uses far fewer multiply operations than either of the LMS algorithms. Also, performing the sign-data adaptation requires only bit shifting multiplies when the step size is a power of two.

Although the performance of the sign-data algorithm as shown in the next figure is quite good, the sign-data algorithm is much less stable than the standard LMS variations. In this noise cancellation example, the signal after processing is a very good match to the input signal, but the algorithm could very easily grow without bound rather than achieve good performance.

Changing coeffs, μ , 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 `adaptfilt.se`

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

$$w(k+1) = w(k) + \mu \operatorname{sgn}[e(k)][x(k)],$$

$$\operatorname{sgn}[e(k)] = \begin{cases} 1, & e(k) > 0 \\ 0, & e(k) = 0 \\ -1, & e(k) < 0 \end{cases}$$

with vector \mathbf{w} containing the weights applied to the filter coefficients and vector \mathbf{x} containing the input data. $e(k)$ (equal to desired signal - filtered signal) is the error at time k and is the quantity the SELMS algorithm seeks to minimize. 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

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

where N is the number of samples in the signal. Also, define mu as a power of two for efficient computation.

Note How you set the initial conditions of the sign-data algorithm profoundly influences the effectiveness of the adaptation. Because the algorithm essentially quantizes the error signal, the algorithm can become unstable easily.

A series of large error values, coupled with the quantization process may result in the error growing beyond all bounds. You restrain the tendency of the sign-error algorithm to get out of control by choosing a small step size ($\mu \ll 1$) and setting the initial conditions for the algorithm to nonzero positive and negative values.

In this noise cancellation example, `adaptfilt.se` requires two input data sets:

- Data containing a signal corrupted by noise. In Using an Adaptive Filter to Remove Noise from an Unknown System on page 4-8, this is $d(k)$, the desired signal. The noise cancellation process removes the noise, leaving the signal.
- Data containing random noise ($x(k)$ in Using an Adaptive Filter to Remove Noise from an Unknown System on page 4-8) that is correlated with the noise that corrupts the signal data. Without the correlation between the noise data, the adapting algorithm cannot remove the noise from the signal.

For the signal, use a sine wave. Note that `signal` is a column vector of 1000 elements.

```
signal = sin(2*pi*0.055*[0:1000-1]');
```

Now, add correlated white noise to `signal`. To ensure that the noise is correlated, pass the noise through a lowpass FIR filter, then add the filtered noise to the signal.

```
noise=randn(1,1000);  
nfilt=fir1(11,0.4); % Eleventh order lowpass filter.  
fnoise=filter(nfilt,1,noise); % Correlated noise data.  
d=signal.'+fnoise;
```

`fnoise` is the correlated noise and `d` is now the desired input to the sign-data algorithm.

To prepare the `adaptfilt` object for processing, set the input conditions `coeffs` and `mu` for the object. As noted earlier in this section, the values you set for `coeffs` and `mu` determine whether the adaptive filter can remove the noise from the signal path. In “System Identification Using `adaptfilt.lms`” on page 4-16, you constructed a default filter that sets the filter coefficients to zeros.

Setting the coefficients to zero often does not work for the sign-error algorithm. The closer you set your initial filter coefficients to the expected values, the more likely it is that the algorithm remains well behaved and converges to a filter solution that removes the noise effectively.

For this example, you start with the coefficients in the filter you used to filter the noise (`nfilt`), and modify them slightly so the algorithm has to adapt.

```
coeffs = nfilt.' -0.01; % Set the filter initial conditions.
mu = 0.05;           % Set step size for algorithm update.
```

With the required input arguments for `adaptfilt.se` prepared, run the adaptation and view the results.

```
ha = adaptfilt.sd(12,mu)
set(ha,'coefficients',coeffs);
set(ha,'persistentmemory',true); % Prevent filter reset.
[y,e] = filter(ha,noise,d);
plot(0:199,signal(1:200),0:199,e(1:200));
```

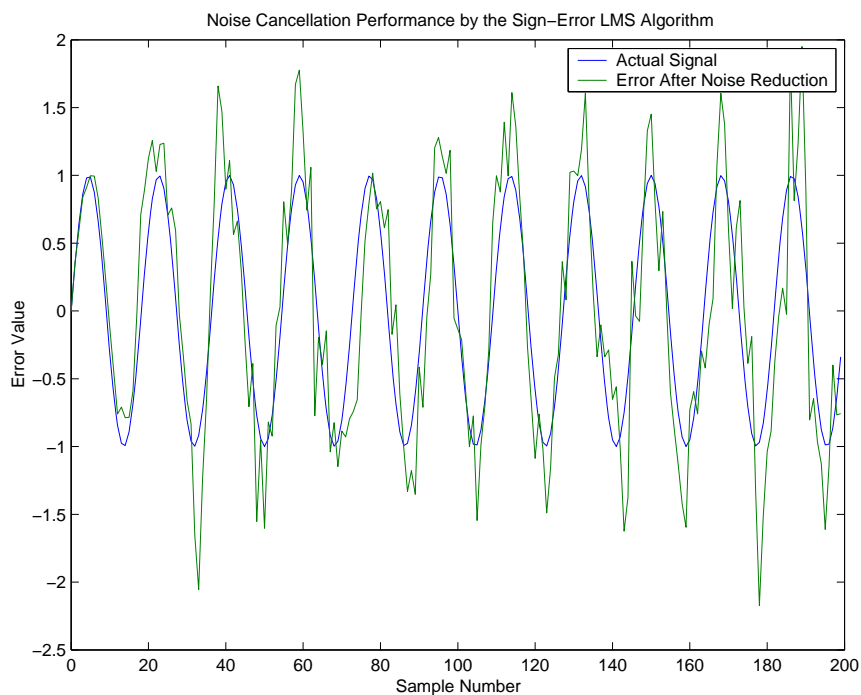
Notice that you have to set the property `PersistentMemory` to `true` when you manually change the settings of object `ha`.

If `PersistentMemory` is left to `false`, the default, when you try to apply `ha` with the method `filter`, the filtering process starts by resetting the object properties to their initial conditions at construction. To preserve the customized coefficients in this example, you set `PersistentMemory` to `true` so the coefficients do not get reset automatically back to zero.

When `adaptfilt.se` runs, it uses far fewer multiply operations than either of the LMS algorithms. Also, performing the sign-error adaptation requires only bit shifting multiplies when the step size is a power of two.

Although the performance of the sign-data algorithm as shown in the next figure is quite good, the sign-data algorithm is much less stable than the standard LMS variations. In this noise cancellation example, the signal after processing is a very good match to the input signal, but the algorithm could very easily become unstable rather than achieve good performance.

Changing `coeffs`, `mu`, or even the lowpass filter you used to create the correlated noise can cause noise cancellation to fail and the algorithm to become useless.



Noise Cancellation Using `adaptfilt.ss`

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 `adaptfilt.sd`” on page 4-22.

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

$$w(k+1) = w(k) + \mu \operatorname{sgn}[e(k)] \operatorname{sgn}[x(k)],$$

$$\operatorname{sgn}[z(k)] = \begin{cases} 1, & z(k) > 0 \\ 0, & z(k) = 0 \\ -1, & z(k) < 0 \end{cases}$$

where

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

Vector \mathbf{w} contains the weights applied to the filter coefficients and vector \mathbf{x} contains the input data. $e(k)$ (= desired signal - filtered signal) is the error at time k and is the quantity the SSLMS algorithm seeks to minimize. μ (μ) is the step size. As you specify μ smaller, the correction to the filter weights gets smaller for each sample and the SSLMS error falls more slowly.

Larger μ 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 μ within the following practical bounds

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

where N is the number of samples in the signal. Also, define μ as a power of two for efficient computation.

Note How you set the initial conditions of the sign-sign algorithm profoundly influences the effectiveness of the adaptation. Because the algorithm essentially quantizes the input signal and the error signal, the algorithm can become unstable easily.

A series of large error values, coupled with the quantization process may result in the error growing beyond all bounds. You restrain the tendency of the sign-sign algorithm to get out of control by choosing a small step size ($\mu \ll 1$) and setting the initial conditions for the algorithm to nonzero positive and negative values.

In this noise cancellation example, `adaptfilt.ss` requires two input data sets:

- Data containing a signal corrupted by noise. In Using an Adaptive Filter to Remove Noise from an Unknown System on page 4-8, this is $d(k)$, the desired signal. The noise cancellation process removes the noise, leaving the cleaned signal as the content of the error signal.
- Data containing random noise ($x(k)$ in Using an Adaptive Filter to Remove Noise from an Unknown System on page 4-8) that is correlated with the noise that corrupts the signal data, called. Without the correlation between the noise data, the adapting algorithm cannot remove the noise from the signal.

For the signal, use a sine wave. Note that `signal` is a column vector of 1000 elements.

```
signal = sin(2*pi*0.055*[0:1000-1]');
```


Now, add correlated white noise to `signal`. To ensure that the noise is correlated, pass the noise through a lowpass FIR filter, then add the filtered noise to the signal.

```
noise=randn(1,1000);
nfilt=fir1(11,0.4); % Eleventh order lowpass filter
fnoise=filter(nfilt,1,noise); % Correlated noise data
d=signal.'+fnoise;
```

`fnoise` is the correlated noise and `d` is now the desired input to the sign-data algorithm.

To prepare the `adaptfilt` object for processing, set the input conditions `coeffs` and `mu` for the object. As noted earlier in this section, the values you set for `coeffs` and `mu` determine whether the adaptive filter can remove the noise from the signal path. In “System Identification Using `adaptfilt.lms`” on page 4-16, you constructed a default filter that sets the filter coefficients to zeros. Usually that approach does not work for the sign-sign algorithm.

The closer you set your initial filter coefficients to the expected values, the more likely it is that the algorithm remains well behaved and converges to a filter solution that removes the noise effectively. For this example, you start with the coefficients in the filter you used to filter the noise (`nfilt`), and modify them slightly so the algorithm has to adapt.

```
coeffs = nfilt.' -0.01; % Set the filter initial conditions.
mu = 0.05; % Set the step size for algorithm updating.
```

With the required input arguments for `adaptfilt.ss` prepared, run the adaptation and view the results.

```
ha = adaptfilt.ss(12,mu)
set(ha,'coefficients',coeffs);
set(ha,'persistentmemory',true); % Prevent filter reset.
[y,e] = filter(ha,noise,d);
plot(0:199,signal(1:200),0:199,e(1:200));
```

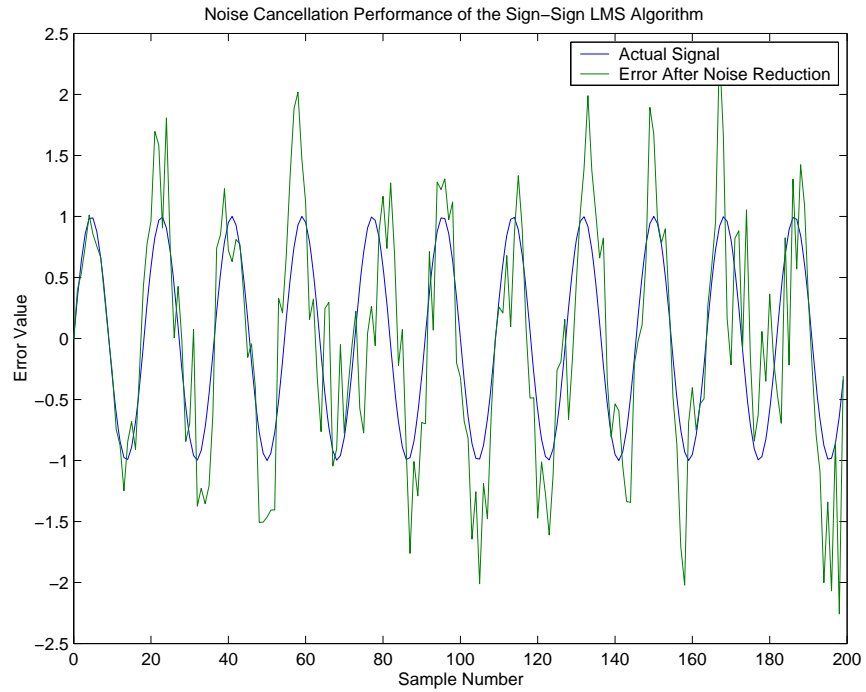
Notice that you have to set the property `PersistentMemory` to `true` when you manually change the settings of object `ha`.

If `PersistentMemory` is left to `false`, when you try to apply `ha` with the method `filter` the filtering process starts by resetting the object properties to their initial conditions at construction. To preserve the customized coefficients in this example, you set `PersistentMemory` to `true` so the coefficients do not get reset automatically back to zero.

When `adaptfilt.ss` runs, it uses far fewer multiply operations than either of the LMS algorithms. Also, performing the sign-sign adaptation requires only bit shifting multiplies when the step size is a power of two.

Although the performance of the sign-sign algorithm as shown in the next figure is quite good, the sign-sign algorithm is much less stable than the standard LMS variations. In this noise cancellation example, the signal after processing is a very good match to the input signal, but the algorithm could very easily become unstable rather than achieve good performance.

Changing `coeffs`, `mu`, or even the lowpass filter you used to create the correlated noise can cause noise cancellation to fail and the algorithm to become useless.



As an aside, the sign-sign LMS algorithm is part of the international CCITT standard for 32 Kb/s ADPCM telephony.

RLS Adaptive Filters

In this section...
“Comparison of the RLS and LMS Adaptive Filter Algorithms” on page 4-36
“Inverse System Identification Using <code>adaptfilt.rls</code> ” on page 4-37

Comparison of the RLS and LMS Adaptive Filter Algorithms

This section provides an introductory example that uses the RLS adaptive filter function `adaptfilt.rls`.

If LMS algorithms represent the simplest and most easily applied adaptive algorithms, the recursive least squares (RLS) algorithms represents increased complexity, computational cost, and fidelity. In performance, RLS approaches the Kalman filter in adaptive filtering applications, at somewhat reduced required throughput in the signal processor.

Compared to the LMS algorithm, the RLS approach offers faster convergence and smaller error with respect to the unknown system, at the expense of requiring more computations.

In contrast to the least mean squares algorithm, from which it can be derived, the RLS adaptive algorithm minimizes the total squared error between the desired signal and the output from the unknown system.

Note that the signal paths and identifications are the same whether the filter uses RLS or LMS. The difference lies in the adapting portion.

Within limits, you can use any of the adaptive filter algorithms to solve an adaptive filter problem by replacing the adaptive portion of the application with a new algorithm.

Examples of the sign variants of the LMS algorithms demonstrated this feature to demonstrate the differences between the sign-data, sign-error, and sign-sign variations of the LMS algorithm.

One interesting input option that applies to RLS algorithms is not present in the LMS processes — a forgetting factor, λ , that determines how the algorithm treats past data input to the algorithm.

When the LMS algorithm looks at the error to minimize, it considers only the current error value. In the RLS method, the error considered is the total error from the beginning to the current data point.

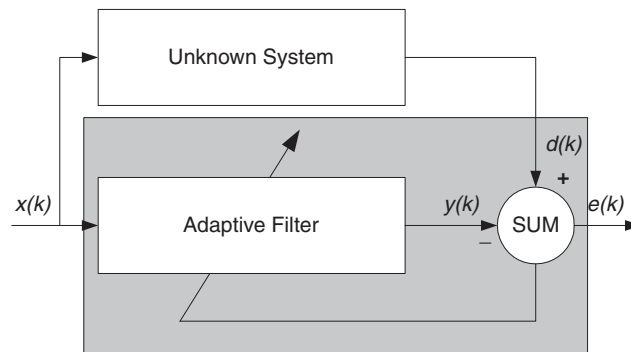
Said another way, the RLS algorithm has infinite memory — all error data is given the same consideration in the total error. In cases where the error value might come from a spurious input data point or points, the forgetting factor lets the RLS algorithm reduce the value of older error data by multiplying the old data by the forgetting factor.

Since $0 \leq \lambda < 1$, applying the factor is equivalent to weighting the older error. When $\lambda = 1$, all previous error is considered of equal weight in the total error.

As λ approaches zero, the past errors play a smaller role in the total. For example, when $\lambda = 0.9$, the RLS algorithm multiplies an error value from 50 samples in the past by an attenuation factor of $0.9^{50} = 5.15 \times 10^{-3}$, considerably deemphasizing the influence of the past error on the current total error.

Inverse System Identification Using `adaptfilt.rls`

Rather than use a system identification application to demonstrate the RLS adaptive algorithm, or a noise cancellation model, this example use the inverse system identification model shown in here.



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

If the transfer function of the unknown is $H(z)$ and the adaptive filter transfer function is $G(z)$, the error measured between the desired signal and the signal from the cascaded system reaches its minimum when the product of $H(z)$ and $G(z)$ is 1, $G(z)*H(z) = 1$. For this relation to be true, $G(z)$ must equal $1/H(z)$, the inverse of the transfer function of the unknown system.

To demonstrate that this is true, create a signal to input to the cascaded filter pair.

```
x = randn(1,3000);
```

In the cascaded filters case, the unknown filter results in a delay in the signal arriving at the summation point after both filters. To prevent the adaptive filter from trying to adapt to a signal it has not yet seen (equivalent to predicting the future), delay the desired signal by 32 samples, the order of the unknown system.

Generally, you do not know the order of the system you are trying to identify. In that case, delay the desired signal by the number of samples equal to half the order of the adaptive filter. Delaying the input requires prepending 12 zero-values samples to x .

```
delay = zeros(1,12);  
d = [delay x(1:2988)]; % Concatenate the delay and the signal.
```

You have to keep the desired signal vector d the same length as x , hence adjust the signal element count to allow for the delay samples.

Although not generally true, for this example you know the order of the unknown filter, so you add a delay equal to the order of the unknown filter.

For the unknown system, use a lowpass, 12th-order FIR filter.

```
ufilt = fir1(12,0.55,'low');
```

Filtering x provides the input data signal for the adaptive algorithm function.

```
xdata = filter(ufilt,1,x);
```

To set the input argument values for the `adaptfilt.rls` object, use the constructor `adaptfilt.rls`, providing the needed arguments `l`, `lambda`, and `invcov`.

For more information about the input conditions to prepare the RLS algorithm object, refer to `adaptfilt.rls` in the reference section of this user's guide.

```
p0 = 2*eye(13);  
lambda = 0.99;  
ha = adaptfilt.rls(13,lambda,p0);
```

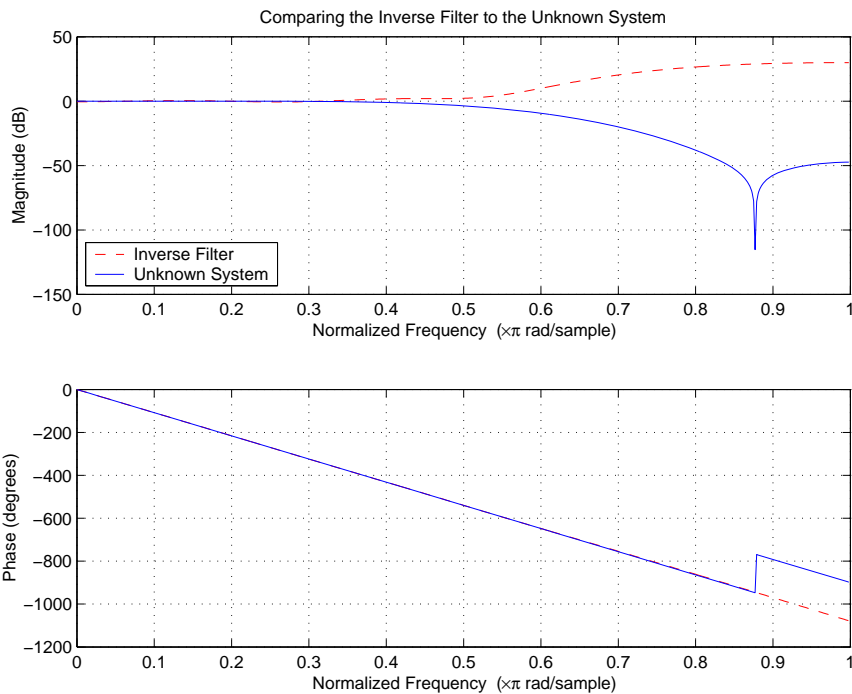
Most of the process to this point is the same as the preceding examples. However, since this example seeks to develop an inverse solution, you need to be careful about which signal carries the data and which is the desired signal.

Earlier examples of adaptive filters use the filtered noise as the desired signal. In this case, the filtered noise (`xdata`) carries the unknown system information. With Gaussian distribution and variance of 1, the unfiltered noise `d` is the desired signal. The code to run this adaptive filter example is

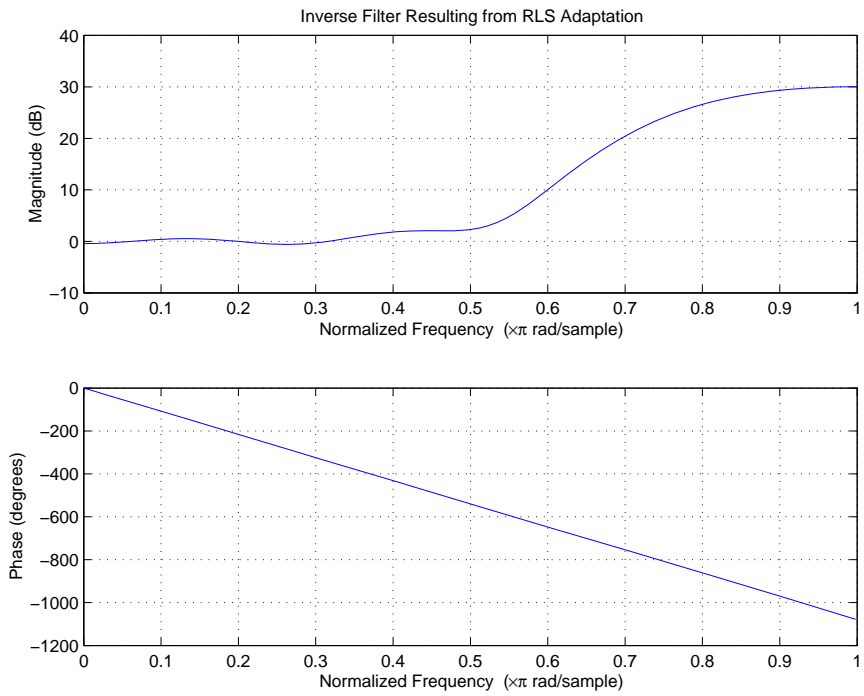
```
[y,e] = filter(ha,xdata,d);
```

where `y` returns the coefficients of the adapted filter and `e` contains the error signal as the filter adapts to find the inverse of the unknown system. You can review the returned elements of the adapted filter in the properties of `ha`.

The next figure presents the results of the adaptation. In the figure, the magnitude response curves for the unknown and adapted filters show. As a reminder, the unknown filter was a lowpass filter with cutoff at 0.55, on the normalized frequency scale from 0 to 1.



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.



Enhance a Signal Using LMS and Normalized LMS Algorithms

In this section...

“Create the Signals for Adaptation” on page 4-42

“Construct Two Adaptive Filters” on page 4-43

“Choose the Step Size” on page 4-44

“Set the Adapting Filter Step Size” on page 4-45

“Filter with the Adaptive Filters” on page 4-45

“Compute the Optimal Solution” on page 4-45

“Plot the Results” on page 4-46

“Compare the Final Coefficients” on page 4-47

“Reset the Filter Before Filtering” on page 4-47

“Investigate Convergence Through Learning Curves” on page 4-48

“Compute the Learning Curves” on page 4-49

“Compute the Theoretical Learning Curves” on page 4-50

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), using `adaptfilt.lms`, and normalized LMS, using `adaptfilt.nlms`, for adaptation.

Create the Signals for Adaptation

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

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

To perform adaptation requires two signals:

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

Generate the Noise Signal

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

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

Corrupt the Desired Signal to Create a Noisy Signal

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

$$x = s + v_1;$$

where s is the desired signal and the noise is v_1 . Adaptive filter processing seeks to recover s from x by removing v_1 . 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 v_2 that is correlated with v_1 . This v_2 is the reference signal for the examples.

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

Construct Two Adaptive Filters

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

```
L = 7;
```

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

Choose the Step Size

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

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

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

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

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

```
mumaxlms =
```

```
0.2096
```

```
mumaxmselms =
```

```
0.1261
```

Set the Adapting Filter Step Size

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

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

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

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

Filter with the Adaptive Filters

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

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

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

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

Compute the Optimal Solution

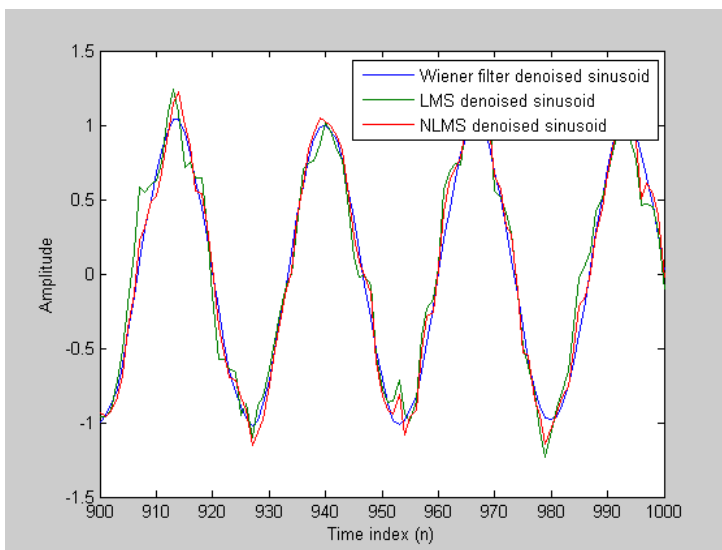
For comparison, compute the optimal FIR Wiener filter.

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

Plot the Results

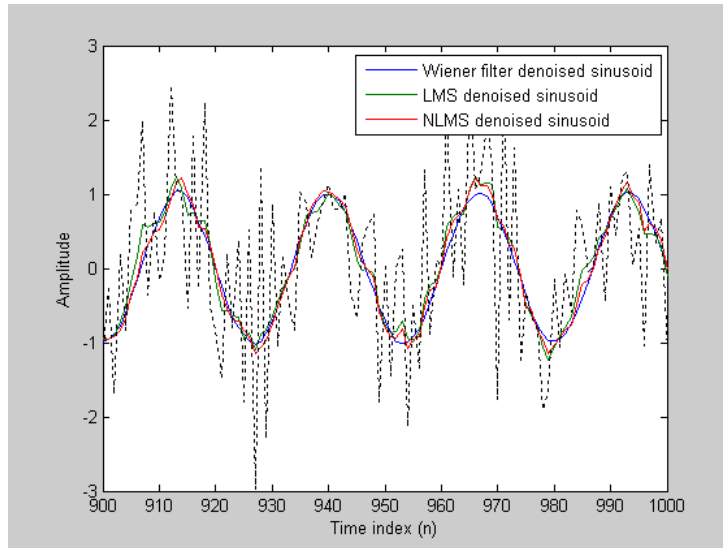
Plot the resulting denoised sinusoid for each filter — the Wiener filter, the LMS adaptive filter, and the NLMS adaptive filter — to compare the performance of the various techniques.

```
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.' h1ms.Coefficients.' hn1ms.Coefficients.']
```

```
ans =
```

```

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

```

Reset the Filter Before Filtering

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

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

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

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

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

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

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

Investigate Convergence Through Learning Curves

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

This demonstration uses 25 sample realizations of the noisy sinusoids.

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

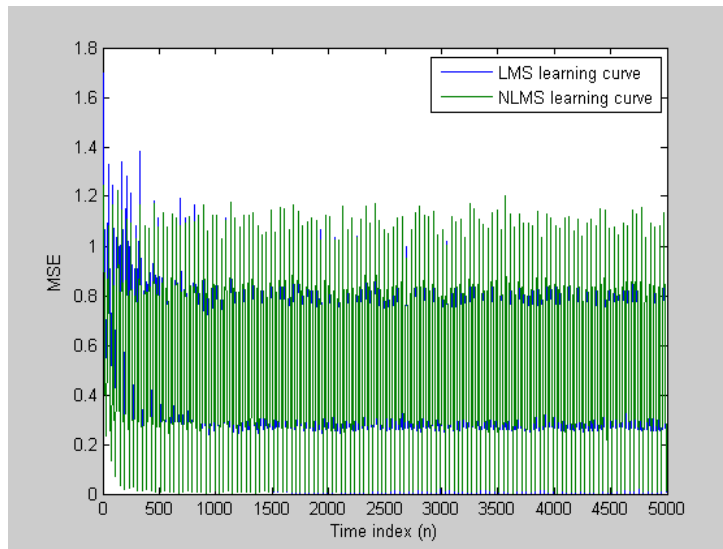

Compute the Learning Curves

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

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

```
reset(hlms);
reset(hnlms);
M = 10; % Decimation factor
mse1ms = msesim(hlms,v2,x,M);
mse1nms = msesim(hnlms,v2,x,M);
plot(1:M:n(end),[mse1ms,mse1nms])
legend('LMS learning curve','NLMS learning curve')
xlabel('Time index (n)');
ylabel('MSE');
```

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

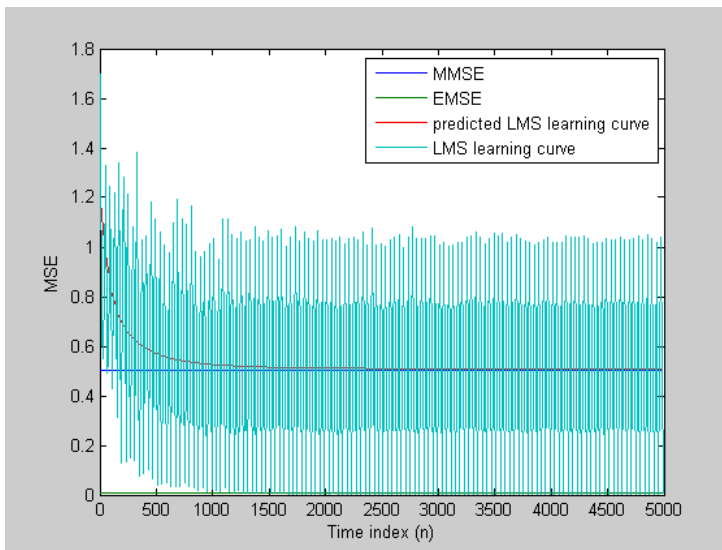


Compute the Theoretical Learning Curves

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

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

```
reset(hlms);
[mmselms,emselms,meanwlms,pmselms] = msepred(hlms,v2,x,M);
plot(1:M:n(end),[mmselms*ones(500,1),emselms*ones(500,1),...
    pmselms,msselms])
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 4-51

“Configure an LMS Filter Block for Adaptive Noise Cancellation” on page 4-53

“Modify Adaptive Filter Parameters During Model Simulation” on page 4-58

“Adaptive Filtering Demos” on page 4-62

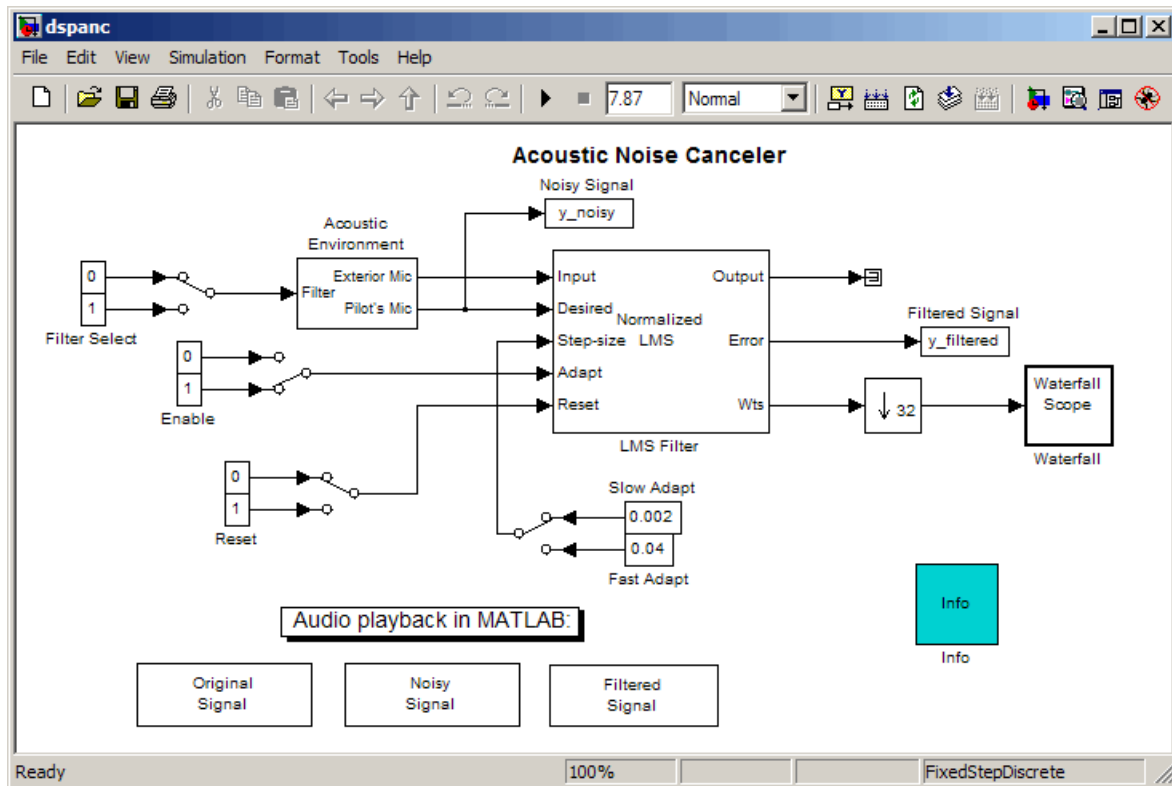
Create an Acoustic Environment in Simulink

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 `ds panc`.

The DSP System Toolbox Acoustic Noise Cancellation demo 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.

Configure an LMS Filter Block for Adaptive Noise Cancellation

In the previous topic, “Create an Acoustic Environment in Simulink” on page 4-51, 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 4-51 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

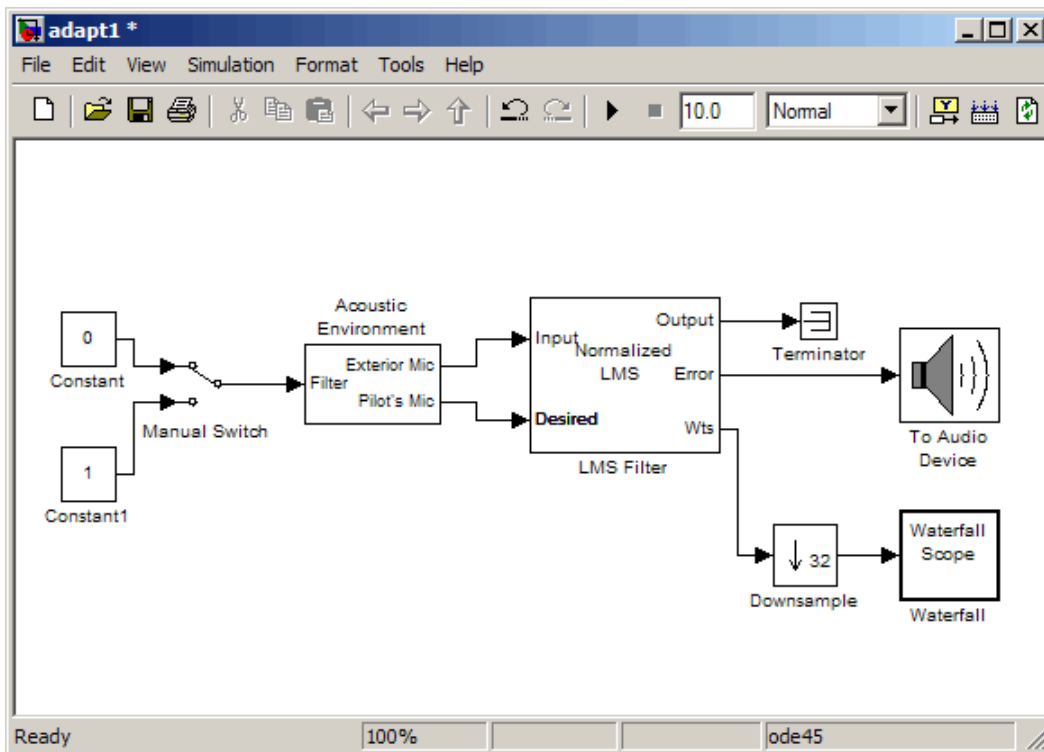
The block uses the normalized LMS algorithm to calculate the forty filter coefficients. Setting the **Leakage factor (0 to 1)** parameter to 1 means that the current filter coefficient values depend on the filter’s initial conditions and all of the previous input values.

- 4 Click-and-drag the following blocks into your model.

4 Adaptive Filters

Block	Library	Quantity
Constant	Simulink/Sources	2
Manual Switch	Simulink/Signal Routing	1
Terminator	Simulink/Sinks	1
Downsample	Signal Operations	1
To Audio Device	Sinks	1
Waterfall Scope	Sinks	1

5 Connect the blocks so that your model resembles the following figure.

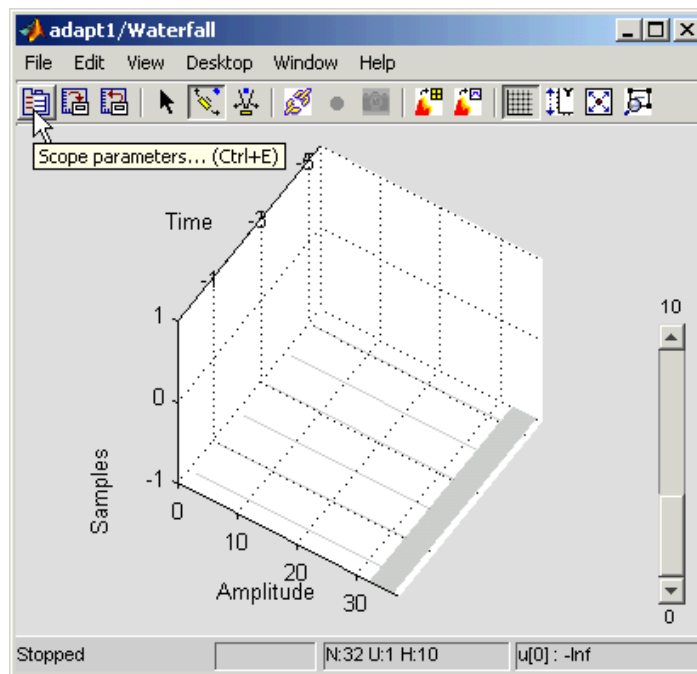


6 Double-click the Constant block. Set the **Constant value** parameter to 0 and then click **OK**.

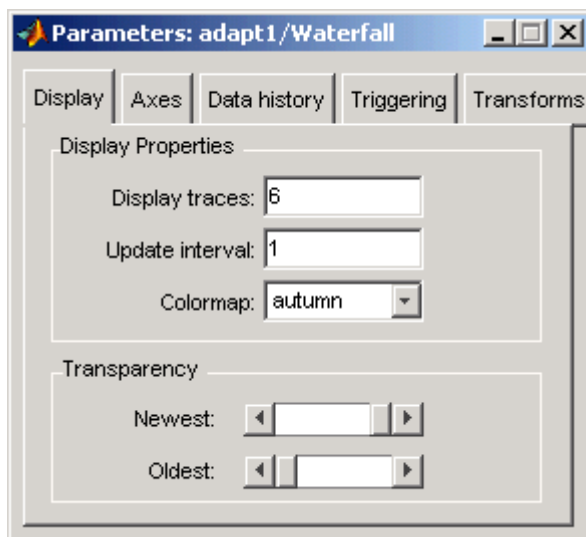
- 7 Double-click the Downsample block. Set the **Downsample factor, K** parameter to 32. Click **OK**.

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.



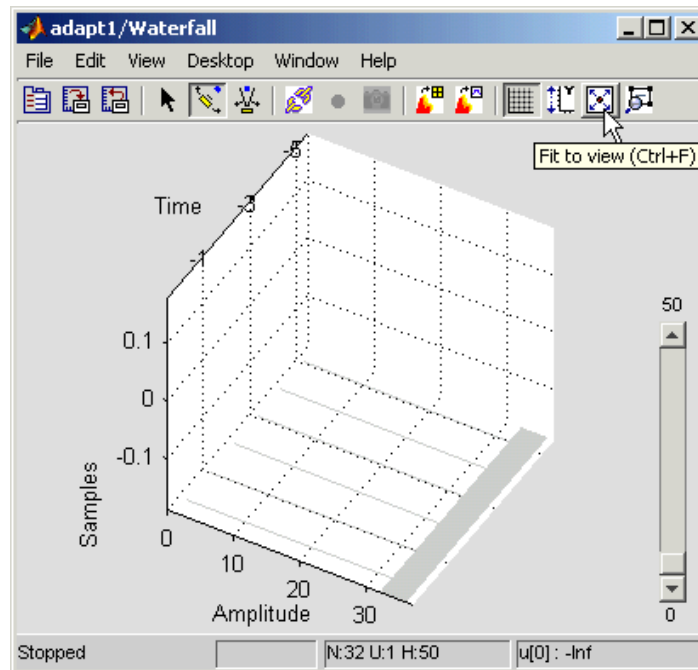
The **Parameters** window opens.



- 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 **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 4-58, you modify the LMS Filter block and change its parameters during simulation.

Modify Adaptive Filter Parameters During Model Simulation

In the previous topic, “Configure an LMS Filter Block for Adaptive Noise Cancellation” on page 4-53, 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:

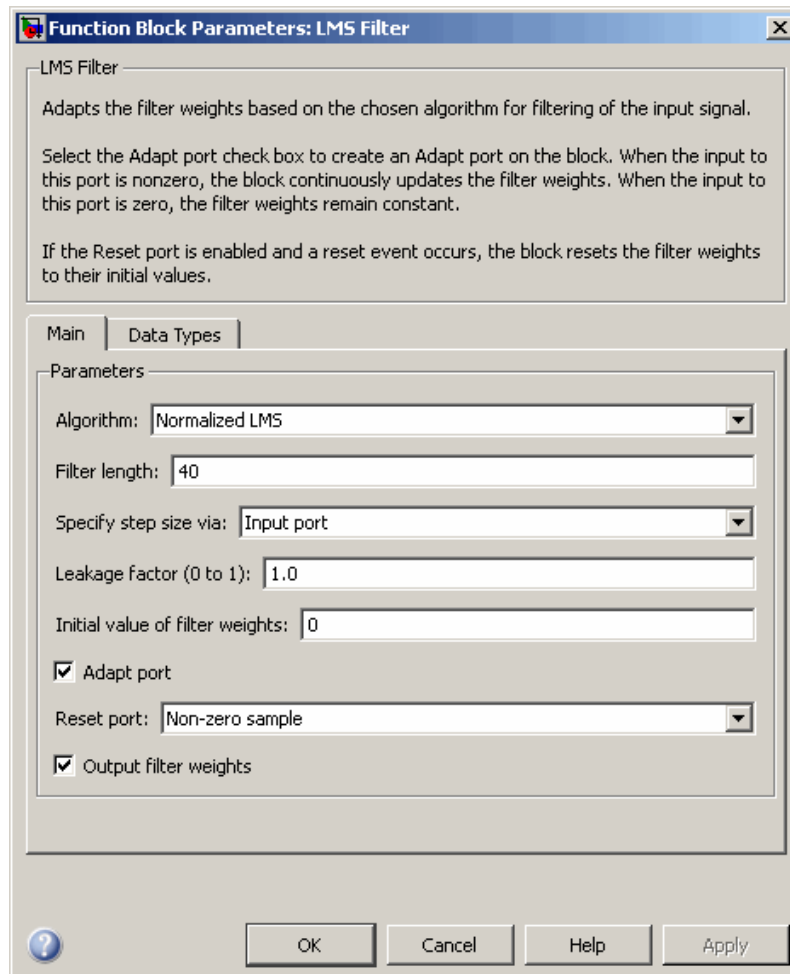
- 1 If the model you created in “Create an Acoustic Environment in Simulink” on page 4-51 is not open on your desktop, you can open an equivalent model by typing

```
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.

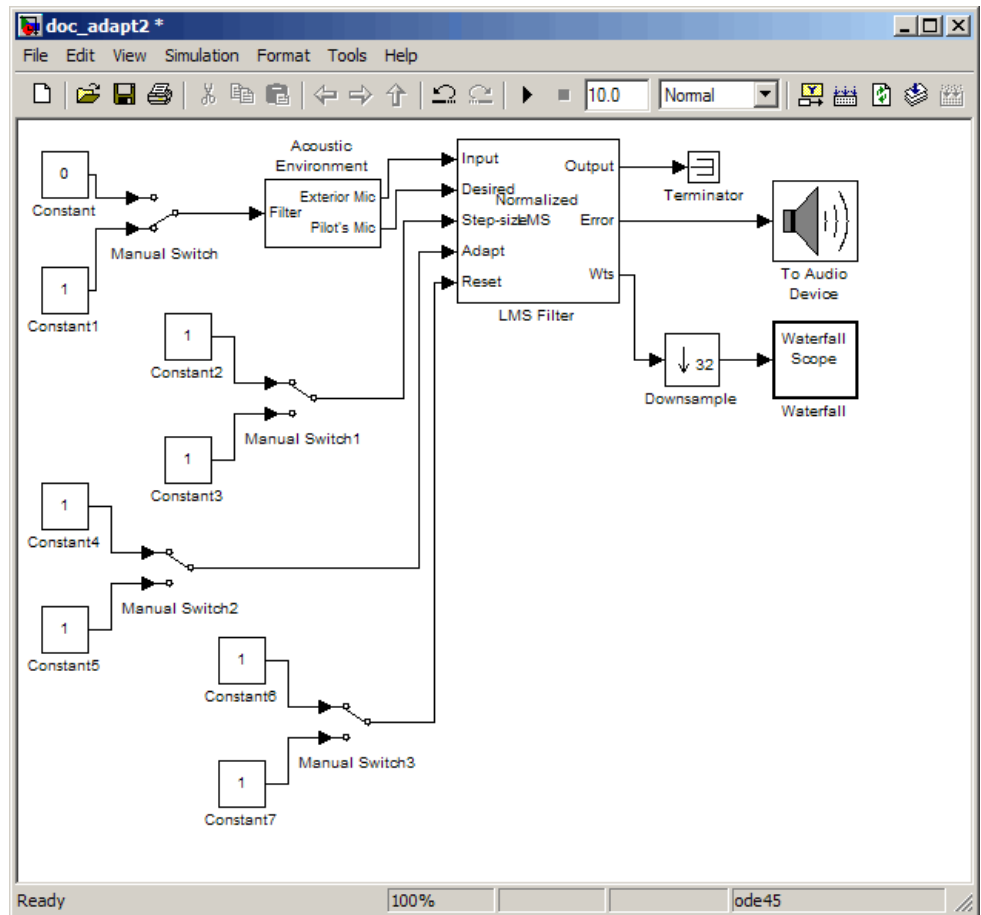


Step-size, Adapt, and Reset ports appear on the LMS Filter block.

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 **Format** menu, point to **Port/Signal Displays**, 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 is 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 Demos

DSP System Toolbox software provides a collection of adaptive filtering demos that illustrate typical applications of the adaptive filtering blocks, listed in the following table.

Adaptive Filtering Demos	Commands for Opening Demos in MATLAB
LMS Adaptive Equalization	<code>lmsadeq</code>
LMS Adaptive Time-Delay Estimation	<code>lmsadtde</code>
Nonstationary Channel Estimation	<code>dspchanest</code>
RLS Adaptive Noise Cancellation	<code>rlsdemo</code>

Opening Demos

To open the adaptive filter demos, click the links in the preceding table in the MATLAB Help browser (not in a Web browser), or type the demo names provided in the table at the MATLAB command line. To access all DSP System Toolbox demos, type `demo toolbox dsp` at the MATLAB command line.

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

Multirate and Multistage Filters

Learn how to analyze, design, and implement multirate and multistage filters in MATLAB and Simulink.

- “Multirate Filters” on page 5-2
- “Multistage Filters” on page 5-6
- “Example Case for Multirate/Multistage Filters” on page 5-8
- “Filter Banks” on page 5-12
- “Examples of Multirate Filtering in Simulink” on page 5-21

Multirate Filters

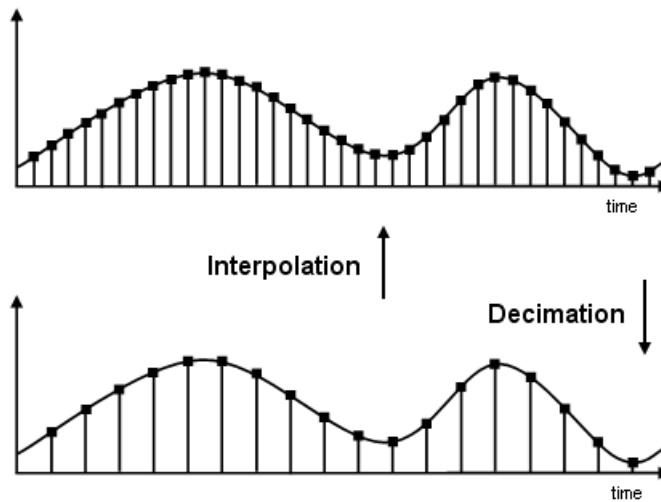
In this section...
“Why Are Multirate Filters Needed?” on page 5-2
“Overview of Multirate Filters” on page 5-2

Why Are Multirate Filters Needed?

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

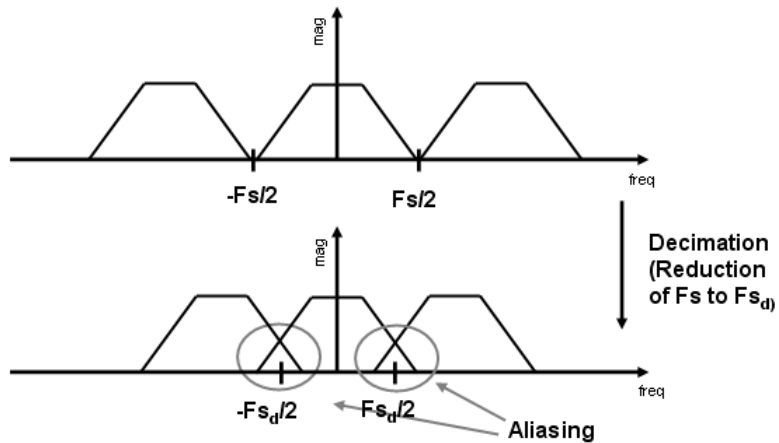
Overview of Multirate Filters

A filter that reduces the input rate is called a *decimator*. A filter that increases the input rate is called an *interpolator*. To visualize this process, examine the following figure, which illustrates the processes of interpolation and decimation in the time domain.

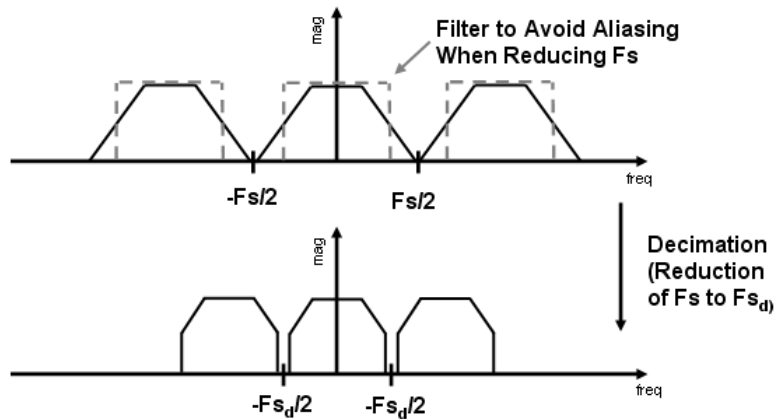


If you start with the top signal, sampled at a frequency F_s , then the bottom signal is sampled at $F_s/2$ frequency. In this case, the decimation factor, or M , is 2.

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 $F_s/2$.

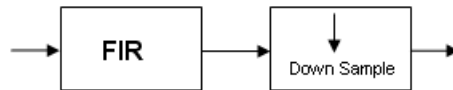


An analogous approach must be taken to avoid *imaging* when performing interpolation on a sampled signal. For more information about the effects of decimation and interpolation on a sampled signal, refer to any one of the references in the Appendix A, “Bibliography” section of the DSP System Toolbox User Guide.

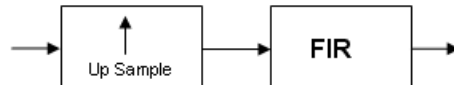
The following list summarizes some guidelines and general requirements regarding decimation and interpolation:

- By the Nyquist Theorem, for band-limited signals, the sampling frequency must be at least twice the bandwidth of the signal. For example, if you have a lowpass filter with the highest frequency of 10 MHz, and a sampling frequency of 60 MHz, the highest frequency that can be handled by the system without aliasing is $60/2=30$, which is greater than 10. You could safely set $M=2$ in this case, since $(60/2)/2=15$, which is still greater than 10.

- If you wish to decimate a signal which does not meet the frequency criteria, you can either:
 - Interpolate first, and then decimate
 - When decimating, you should apply the filter first, then perform the decimation. When interpolating a signal, you should interpolate first, then filter the signal.
- Typically in decimation of a signal a filter is applied first, thereby allowing decimation without aliasing, as shown in the following figure:



- Conversely, a filter is typically applied after interpolation to avoid imaging:



- M must be an integer. Although, if you wish to obtain an M of $4/5$, you could interpolate by 4, and then decimate by 5, provided that frequency restrictions are met. This type of multirate filter will be referred to as a *sample rate converter* in the documentation that follows.

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

Multistage Filters

In this section...
“Why Are Multistage Filters Needed?” on page 5-6
“Optimal Multistage Filters in DSP System Toolbox Software” on page 5-6

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 multistage 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 multirate system, where there is a decimator or an interpolator with a large factor. In these cases, it is usually wise to break up the filter into several multirate stages, each comprising a multiple of the total decimation/interpolation factor.

Optimal Multistage Filters in DSP System Toolbox Software

As described in the previous section, within a multirate filter each interconnected filter is called a *stage*. While it is possible to design a multistage filter manually, it is also possible to perform automatic optimization of a multistage filter automatically. When designing a filter manually it can be difficult to guess how many stages would provide an optimal design, optimize each stage, and then optimize all the stages together. 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” on page 5-8

“Single-Rate/Single-Stage Equiripple Design” on page 5-8

“Reduce Computational Cost Using Multirate/Multistage Design” on page 5-9

“Compare the Responses” on page 5-9

“Further Performance Comparison” on page 5-10

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 demo entitled Efficient Narrow Transition-Band FIR Filter Design.

Single-Rate/Single-Stage Equiripple Design

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

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

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

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

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


When you determine the cost of this design, you can see that 695 multipliers are required.

```
cost(Hd)
```

Reduce Computational Cost Using Multirate/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');
```

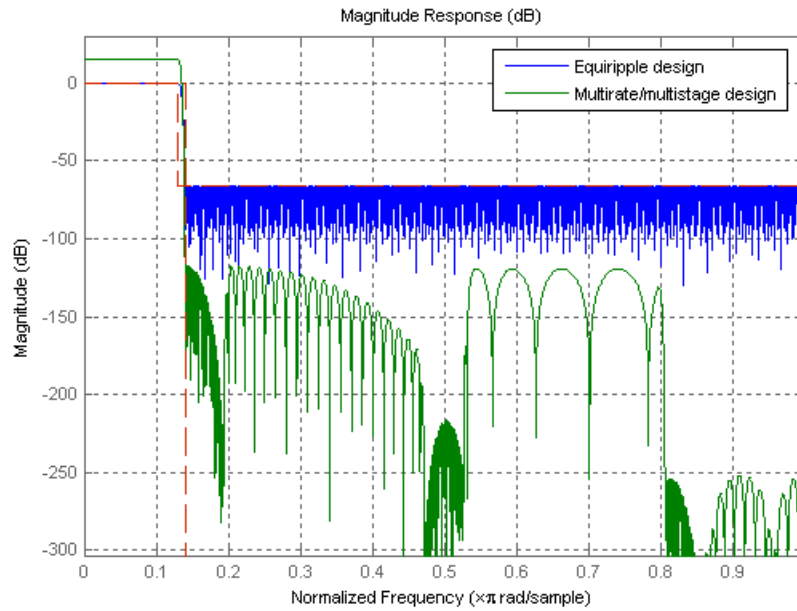
You can then view the cost of the filter created using this design step, and you can see that a significant cost advantage has been achieved.

```
cost(Hd_multi)
```

Compare the Responses

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

```
hfvtool(Hd, Hd_multi);  
legend(hfvtool, '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

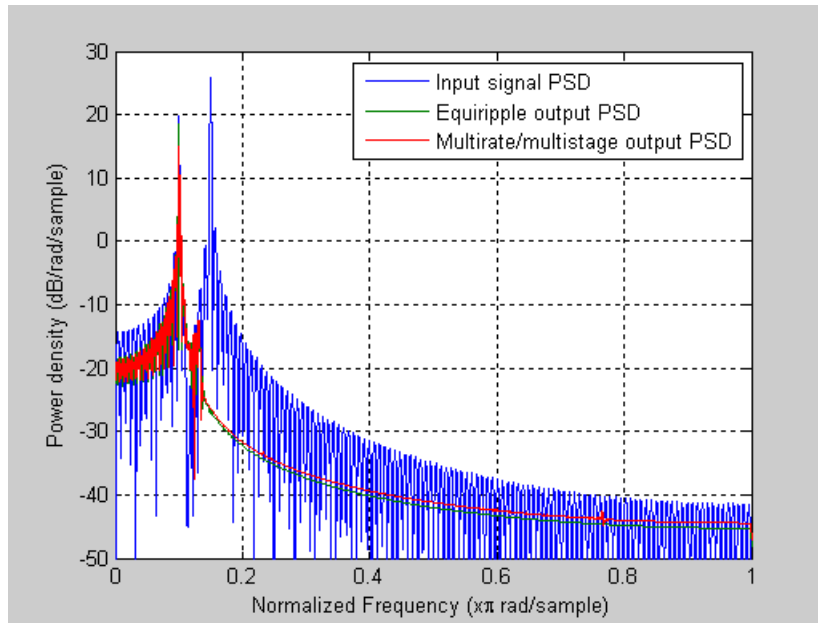
You can check the performance of the multirate/multistage design by plotting the power spectral densities of the input and the various outputs, and you can see that the sinusoid at 0.4π is attenuated comparably by both the equiripple design and the multirate/multistage design.

```
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

```



Filter Banks

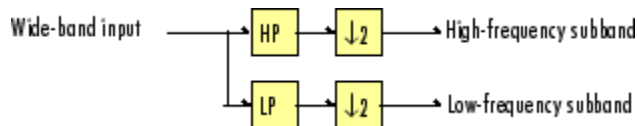
Multirate filters alter the sample rate of the input signal during the filtering process. Such filters are useful in both rate conversion and filter bank applications.

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

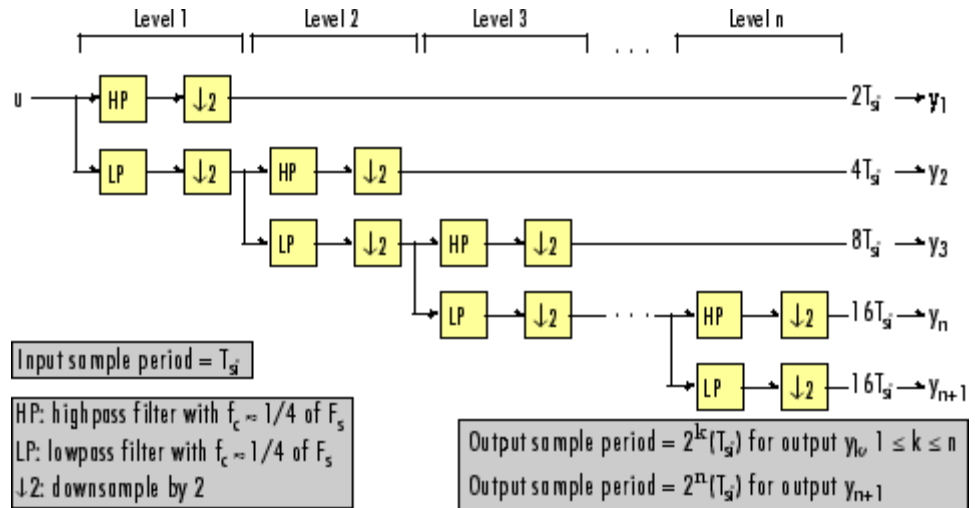
Dyadic analysis filter banks are constructed from the following basic unit. The unit can be cascaded to construct dyadic analysis filter banks with either a symmetric or asymmetric tree structure.



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.

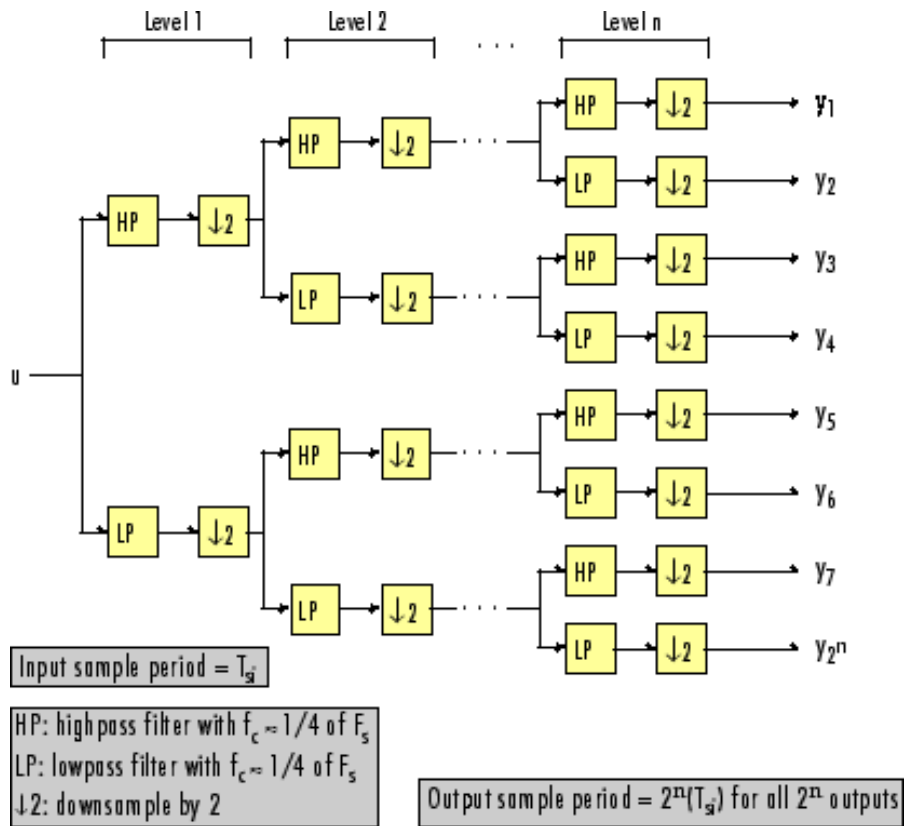
The unit decomposes its input into adjacent high-frequency and low-frequency subbands. Compared to the input, each subband has half the bandwidth (due to the half-band filters) and half the sample rate (due to the decimation by 2).

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

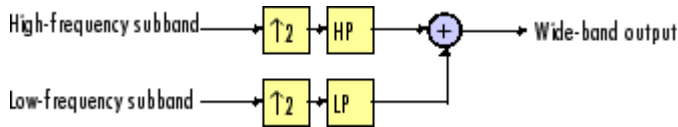
Characteristic	N-Level Symmetric	N-Level Asymmetric
Low- and High-Frequency Subband Decomposition	All the low-frequency and high-frequency subbands in a level are decomposed in the next level.	Each level's low-frequency subband is decomposed in the next level, and each level's high-frequency band is 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 $BW_k = \begin{cases} BW / 2^k & (1 \leq k \leq n) \\ BW / 2^n & (k = n + 1) \end{cases}$ $N_k = \begin{cases} N / 2^k & (1 \leq k \leq n) \\ N / 2^n & (k = n + 1) \end{cases}$ <p>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.</p>

Notable Characteristics of Asymmetric and Symmetric Dyadic Analysis Filter Banks (Continued)

Characteristic	N-Level Symmetric	N-Level Asymmetric
Output Sample Period	All output subbands have a sample period of $2^n(T_{si})$	Sample period of kth output $= \begin{cases} 2^k(T_{si}) & (1 \leq k \leq n) \\ 2^n(T_{si}) & (k = n + 1) \end{cases}$ <p>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.</p>
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

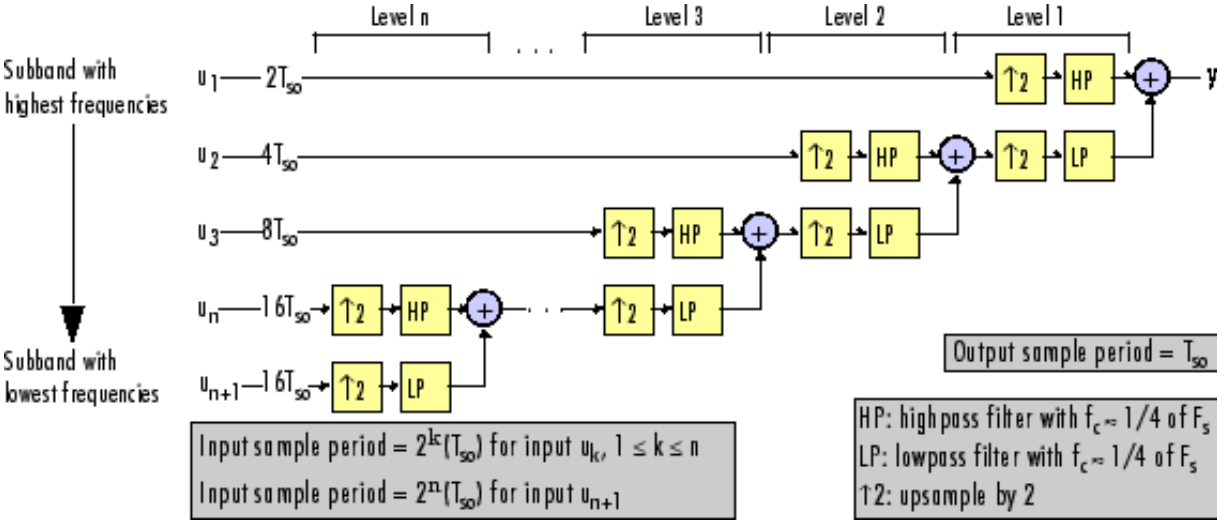
Dyadic synthesis filter banks are constructed from the following basic unit. The unit can be cascaded to construct dyadic synthesis filter banks with either a asymmetric or symmetric tree structure as illustrated in the figures entitled n-Level Asymmetric Dyadic Synthesis Filter Bank and n-Level Symmetric 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.

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.
Number of Input Subbands	2^n	$n+1$
Bandwidth and Number of Samples in Input Subbands	All input 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 k th input subband has the following bandwidth and number of samples. $BW_k = \begin{cases} BW / 2^k & (1 \leq k \leq n) \\ BW / 2^n & (k = n + 1) \end{cases}$ $N_k = \begin{cases} N / 2^k & (1 \leq k \leq n) \\ N / 2^n & (k = n + 1) \end{cases}$
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 k th input subband $= \begin{cases} 2^k(T_{so}) & (1 \leq k \leq n) \\ 2^n(T_{so}) & (k = n + 1) \end{cases}$ where the output sample period is T_{so} .

Notable Characteristics of Asymmetric and Symmetric Dyadic Synthesis Filter Banks (Continued)

Characteristic	N-Level Symmetric	N-Level Asymmetric
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 <i>analysis</i> filter bank is exactly canceled in the reconstruction of the signal in the dyadic <i>synthesis</i> filter bank.	

For more information, see Dyadic Synthesis Filter Bank.

Examples of Multirate Filtering in Simulink

DSP System Toolbox software provides a collection of multirate filtering demos and example models that illustrate typical applications of the multirate filtering blocks. To open the demos and example models, click on the links in the following tables in the MATLAB Help browser (not in a Web browser), or type the names provided at the MATLAB command line. To access all DSP System Toolbox demos, type `demo toolbox dsp` at the MATLAB command line.

Multirate Filtering Demos	Description	Command for Opening Demos 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	<code>dspaudiosrc</code>
Sigma-Delta A/D Converter	Illustrates analog-to-digital conversion using a sigma-delta algorithm implementation	<code>dspsdadc</code>
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	<code>dspwavelet</code>

Multirate Filtering Example Models	Description	Command for Opening Example Models in MATLAB
Frame-Based Narrowband Bandpass Filter	Uses FIR Decimation and Interpolation blocks in multiples stages to create a frame-based narrowband bandpass filter with low computational load	<code>ex_mrf_nbpf</code>
Frame-Based Narrowband Highpass Filter	Uses FIR Decimation and Interpolation blocks in multiples stages to create a frame-based narrowband highpass filter with low computational load	<code>ex_mrf_nhpf</code>

Multirate Filtering Example Models	Description	Command for Opening Example Models in MATLAB
Frame-Based Narrowband Lowpass Filter	Uses FIR Decimation and Interpolation blocks in multiples stages to create a frame-based narrowband lowpass filter with low computational load	ex_mrf_nlpf
Frame-Based Wideband Highpass Filter	Uses FIR Decimation and Interpolation blocks in multiples stages to create a frame-based wideband highpass filter with low computational load	ex_mrf_whpf
Frame-Based Wideband Lowpass Filter	Uses FIR Decimation and Interpolation blocks in multiples stages to create a frame-based wideband lowpass filter with low computational load	ex_mrf_wlpf
Sample-Based Narrowband Bandpass Filter	Uses FIR Decimation and Interpolation blocks in multiples stages to create a sample-based narrowband bandpass filter with low computational load	ex_mrf_nbp
Sample-Based Narrowband Highpass Filter	Uses FIR Decimation and Interpolation blocks in multiples stages to create a sample-based narrowband highpass filter with low computational load	ex_mrf_nhp
Sample-Based Narrowband Lowpass Filter	Uses FIR Decimation and Interpolation blocks in multiples stages to create a sample-based narrowband lowpass filter with low computational load	ex_mrf_nlp
Sample-Based Wideband Highpass Filter	Uses FIR Decimation and Interpolation blocks in multiples stages to create a sample-based wideband highpass filter with low computational load	ex_mrf_whp
Sample-Based Wideband Lowpass Filter	Uses FIR Decimation and Interpolation blocks in multiples stages to create a sample-based wideband lowpass filter with low computational load	ex_mrf_wlp

Transforms, Estimation, and Spectral Analysis

Learn about transforms, estimation and spectral analysis.

- “Transform Time-Domain Data into the Frequency Domain Using the FFT Block” on page 6-2
- “Transform Frequency-Domain Data into the Time Domain Using the IFFT Block” on page 6-7
- “Linear and Bit-Reversed Output Order” on page 6-12
- “Calculate the Channel Latencies Required for Wavelet Reconstruction” on page 6-14
- “Spectral Analysis” on page 6-23
- “Power Spectrum Estimates” on page 6-24
- “Spectrograms” on page 6-34

Transform Time-Domain Data into the Frequency Domain Using the FFT Block

When you want to transform time-domain data into the frequency domain, use the FFT block. You can find additional background information on transform operations in the “Signal Processing Toolbox” documentation.

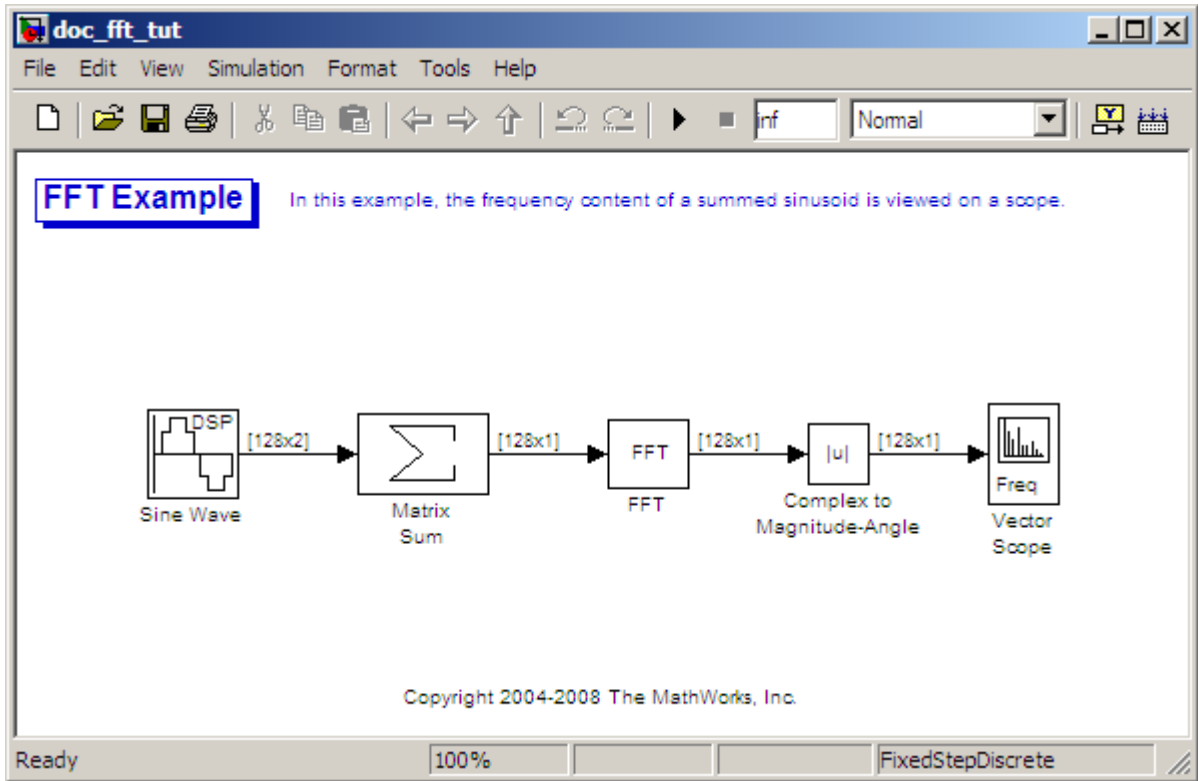
In this example, you use the Sine Wave block to generate two sinusoids, one at 15 Hz and the other at 40 Hz. You sum the sinusoids point-by-point to generate the compound sinusoid

$$u = \sin(30\pi t) + \sin(80\pi t)$$

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.

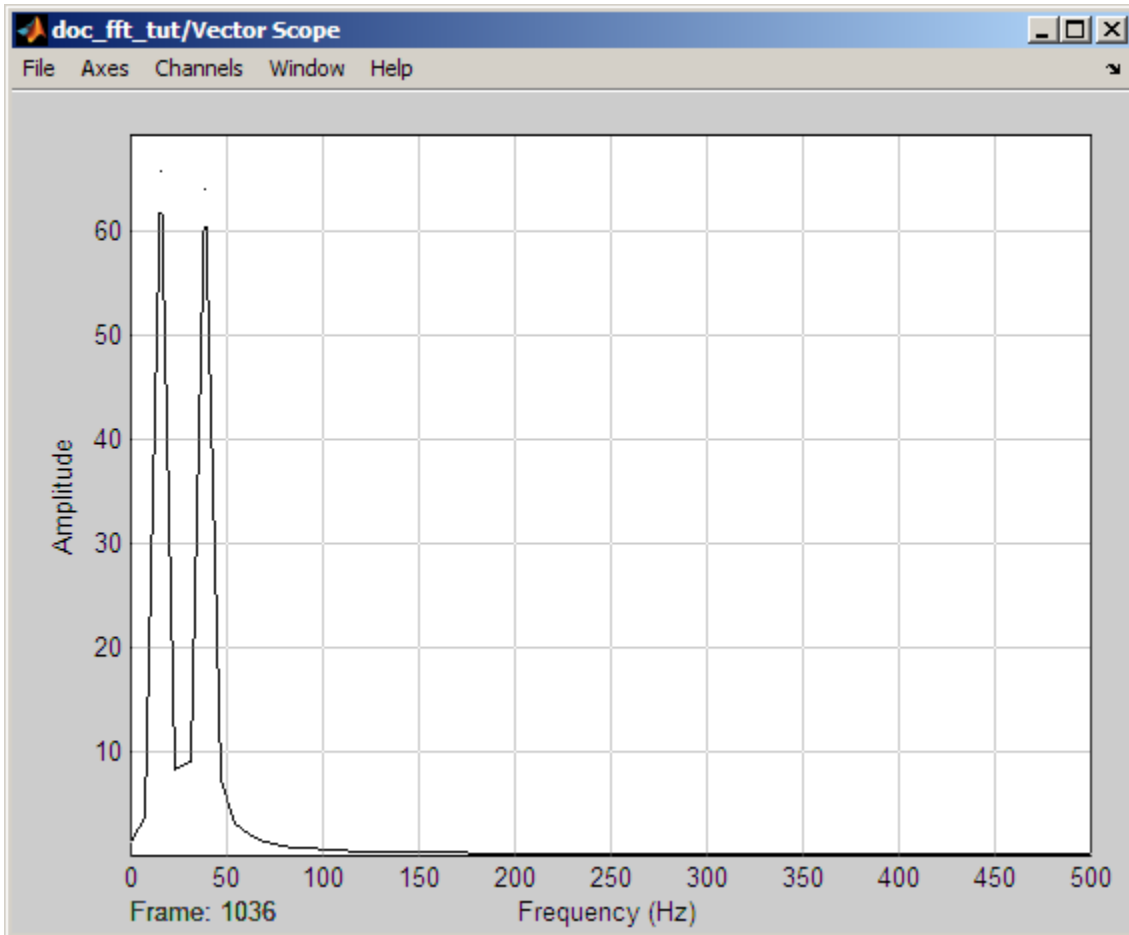
- 7 Double-click the Complex to Magnitude-Angle block. The **Block Parameters: Complex to Magnitude-Angle** dialog box opens.
- 8 Set the **Output** parameter to **Magnitude**, and then click **OK**.

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** = $[0 \dots F_s/2]$
 - 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 Scope block, which computes the magnitude FFT internally. Other blocks that compute the FFT

internally are the blocks in the Power Spectrum Estimation library. See “Spectral Analysis” on page 6-23 for more information about these blocks.

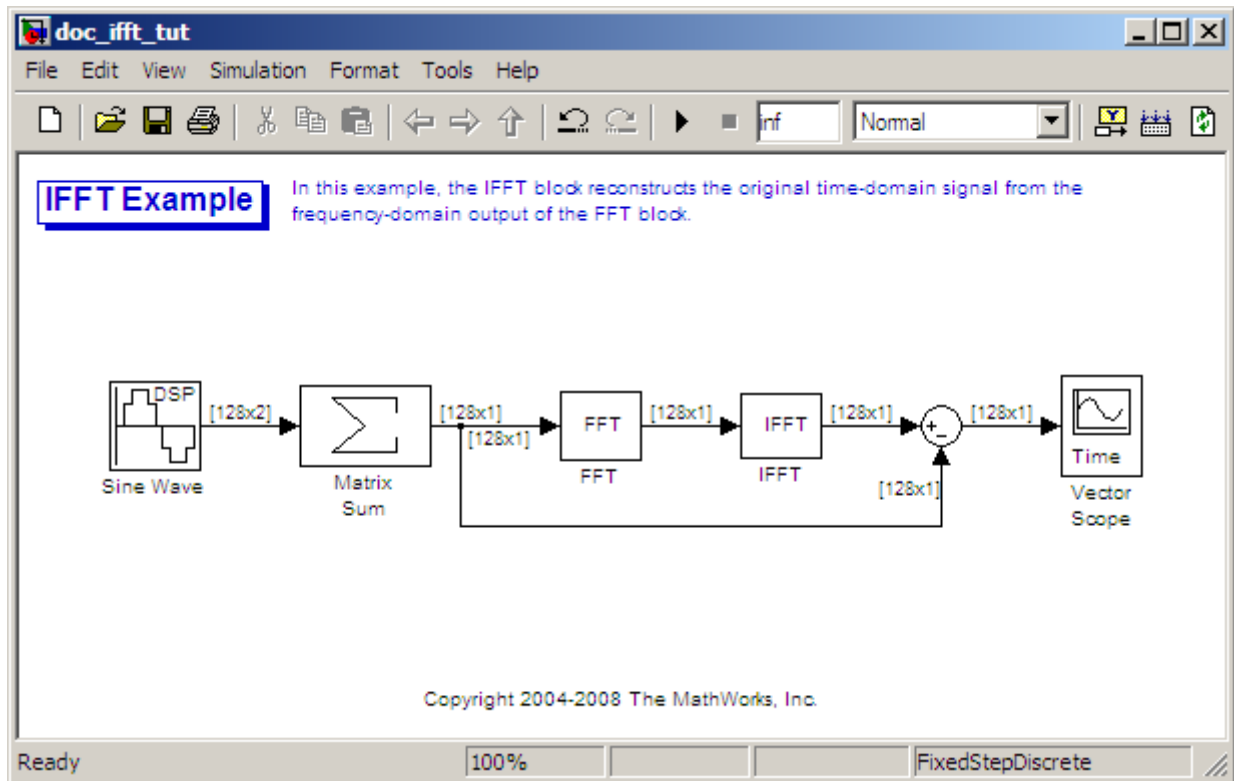
Transform Frequency-Domain Data into the Time Domain Using the IFFT Block

When you want to transform frequency-domain data into the time domain, use the IFFT block. You can find additional background information on transform operations in the “Signal Processing Toolbox” documentation.

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.

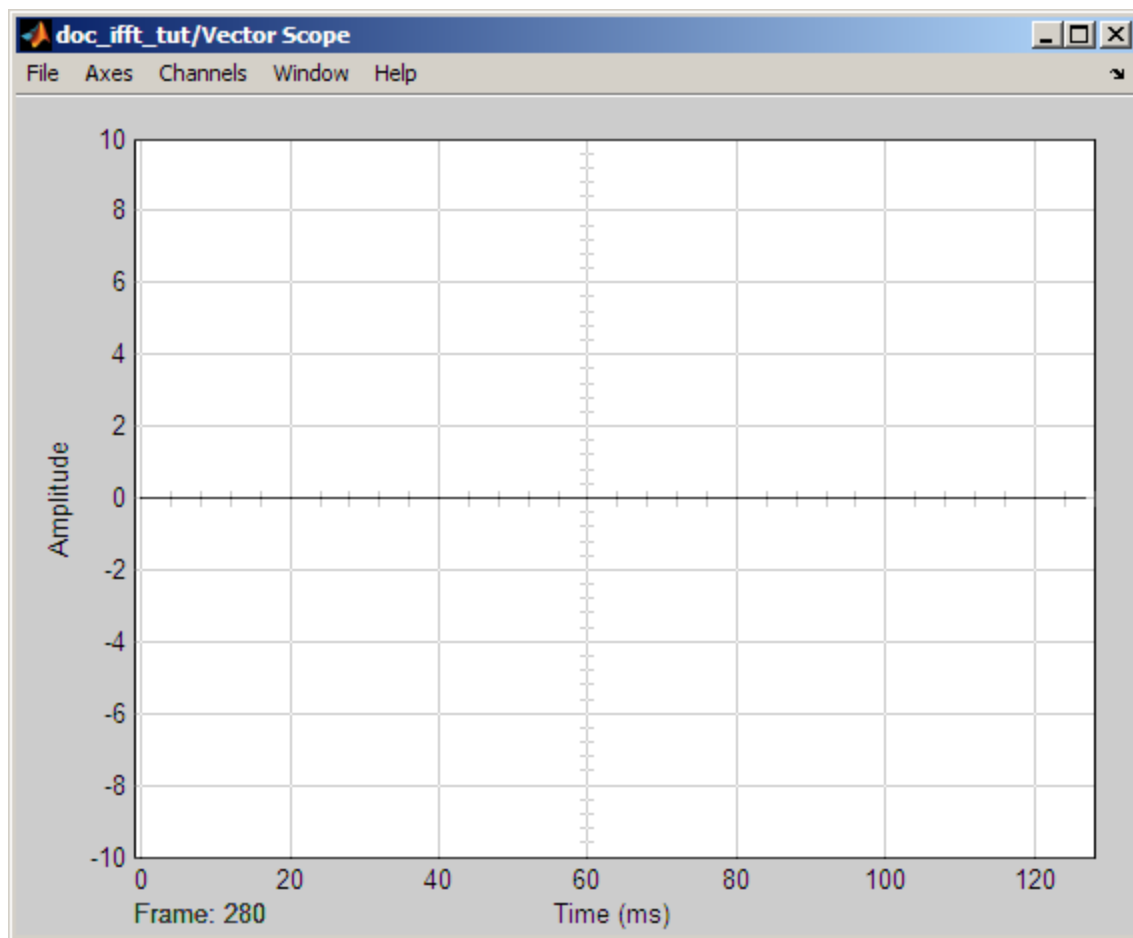
- 7 Double-click the FFT block. The **Block Parameters: FFT** dialog box opens.
- 8 Select the **Output in bit-reversed order** check box., and then click **OK**.
- 9 Double-click the IFFT block. The **Block Parameters: IFFT** dialog box opens.
- 10 Set the block parameters as follows, and then click **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.

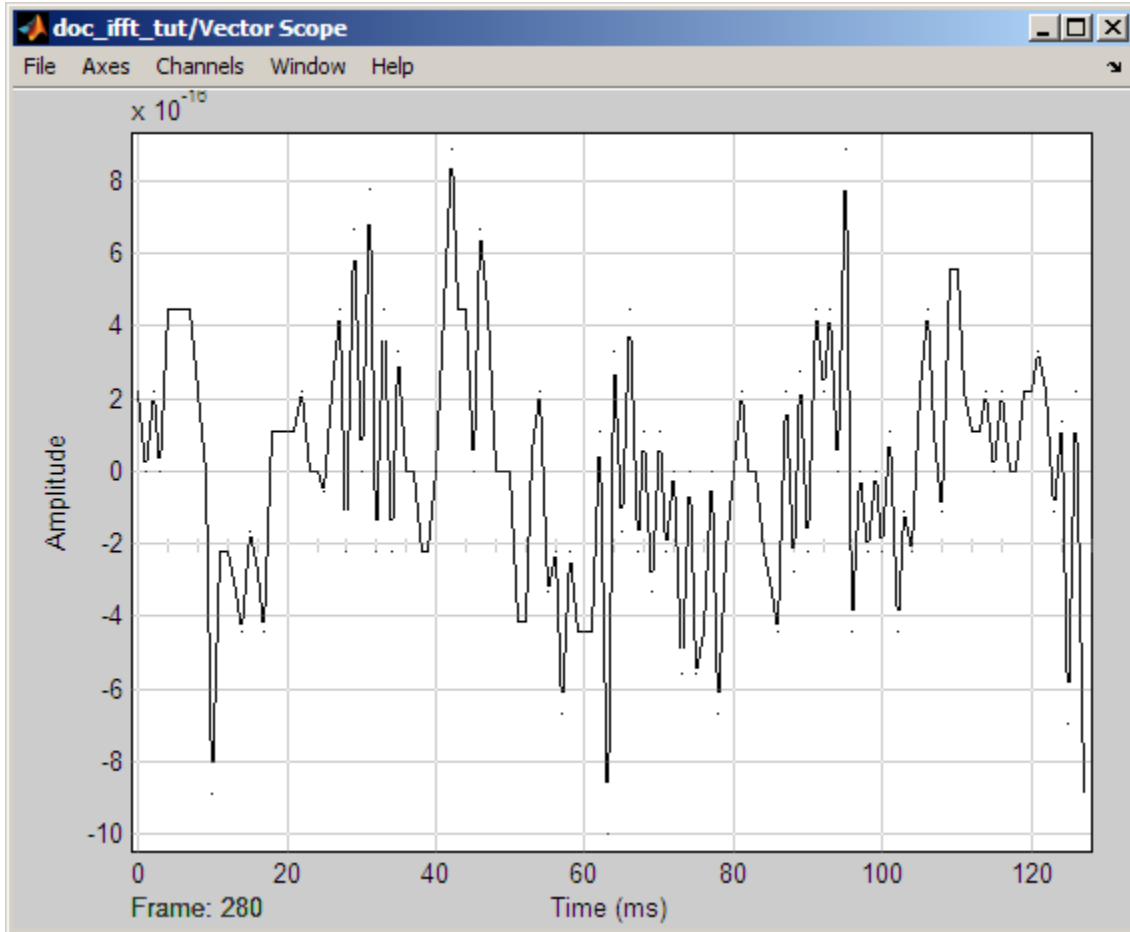
- 11 Double-click the Vector Scope block.
- 12 Set the block parameters as follows, and then click **OK**:
 - Click the **Scope Properties** tab.
 - **Input domain** = Time

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 6-12
“Find the Bit-Reversed Order of Your Frequency Indices” on page 6-12

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.

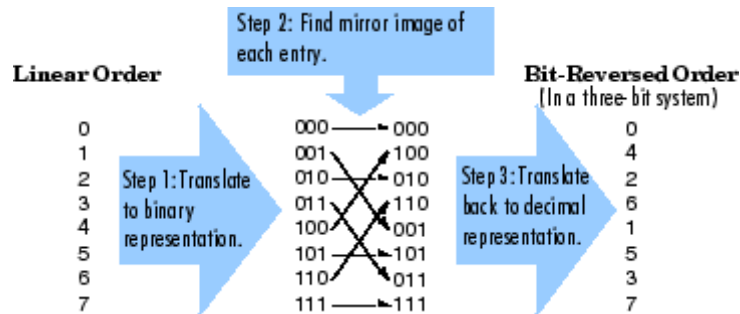
The input to the IFFT block can be in linear or bit-reversed order. Therefore, you do not have to alter the ordering of your data before transforming it back into the time domain. However, the IFFT block may run more quickly when the input is provided in bit-reversed order.

Find the Bit-Reversed Order of Your Frequency Indices

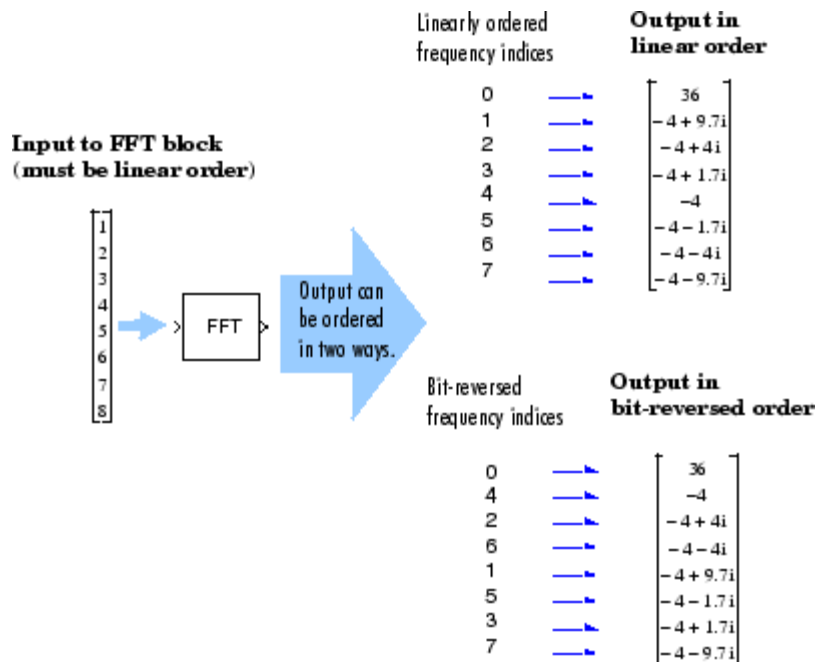
Two numbers are bit-reversed values of each other when the binary representation of one is the mirror image of the binary representation of the other. For example, in a three-bit system, one and four are bit-reversed values of each other, since the three-bit binary representation of one, 001, is the 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.



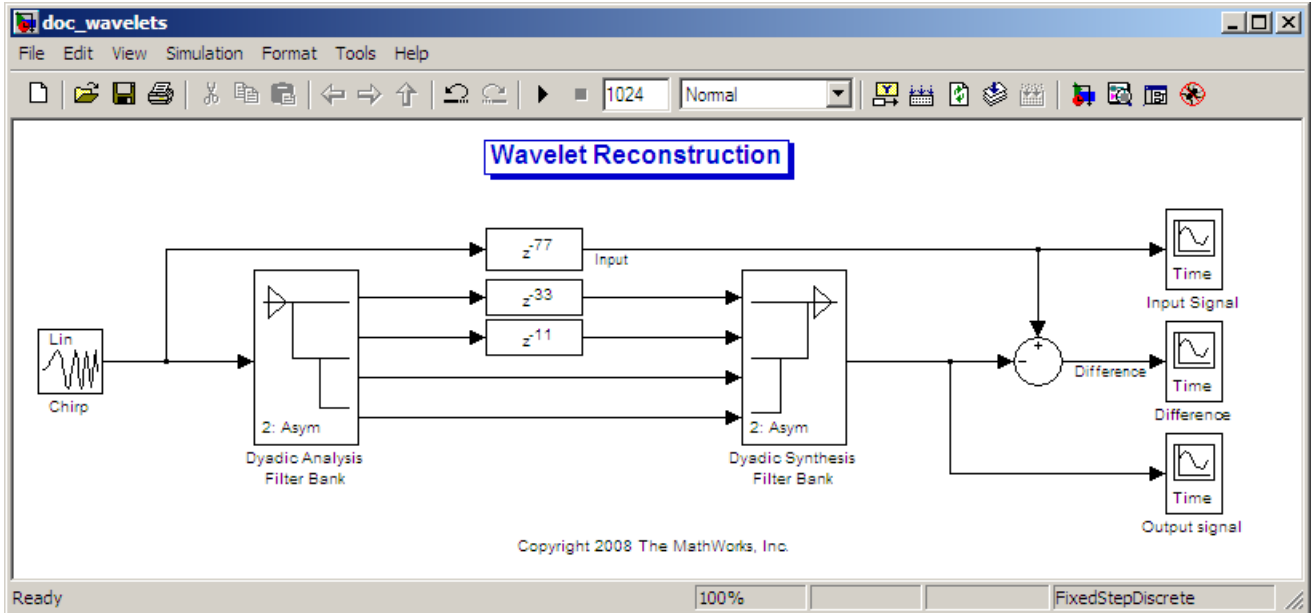
Calculate the Channel Latencies Required for Wavelet Reconstruction

In this section...
“Analyze Your Model” on page 6-14
“Calculate the Group Delay of Your Filters” on page 6-16
“Reconstruct the Filter Bank System” on page 6-18
“Equalize the Delay on Each Filter Path” on page 6-18
“Update and Run the Model” on page 6-21
“References” on page 6-22

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:

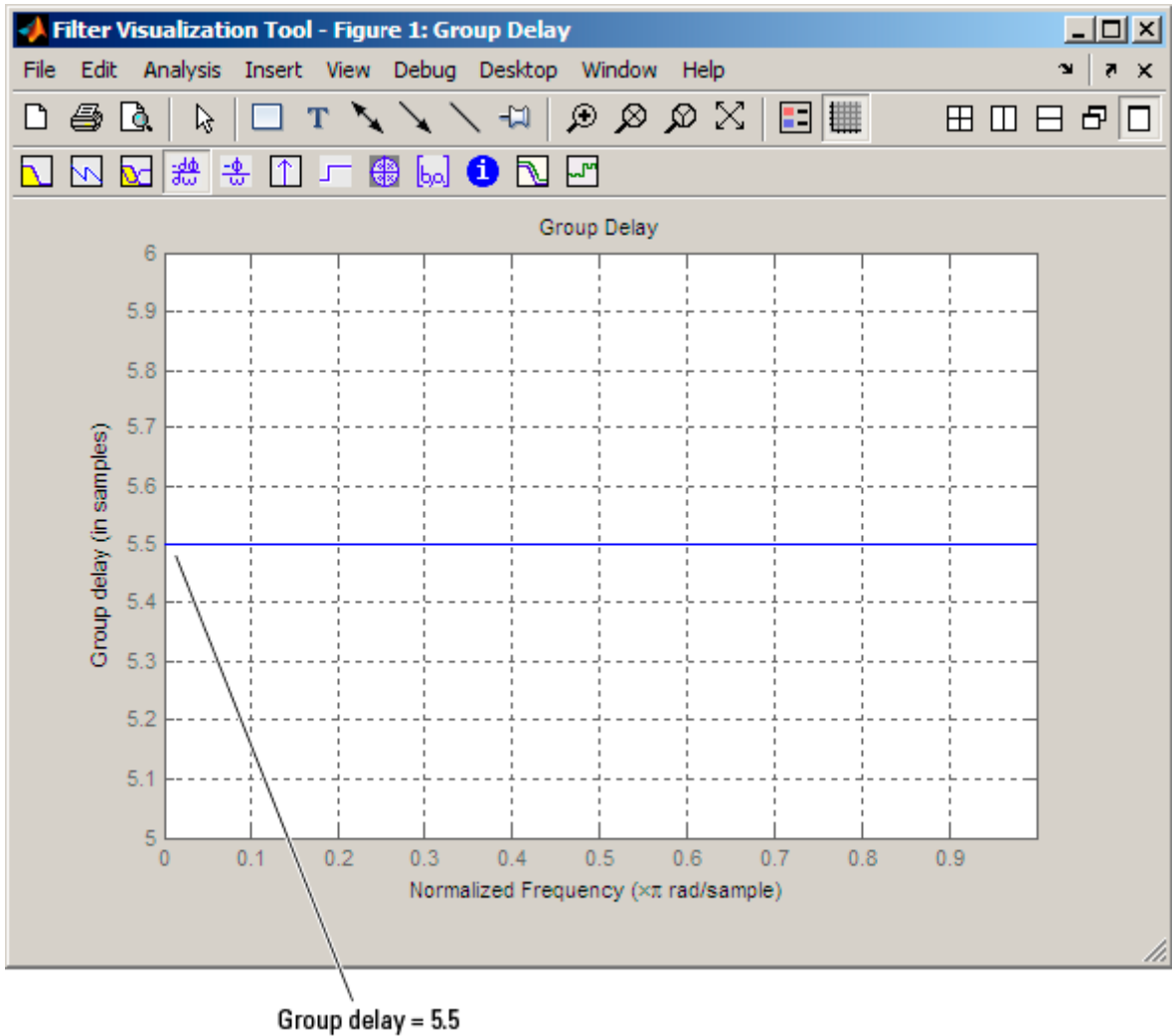
```
[Lo_D, Hi_D, Lo_R, Hi_R] = wfilters('bior3.5')
```

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.



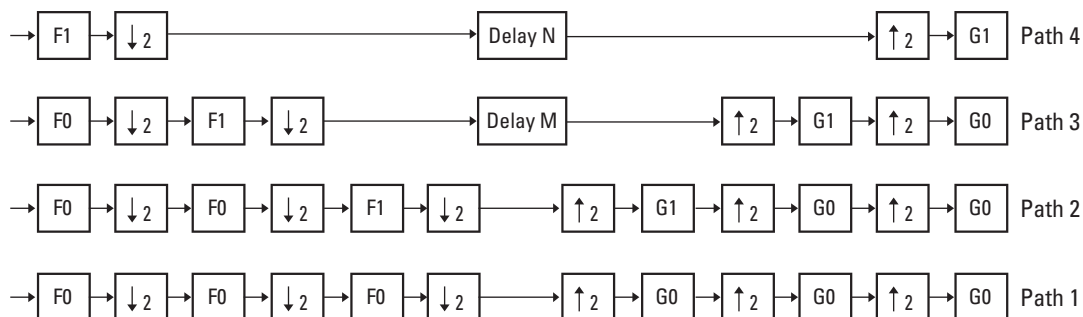
Note Repeat this procedure to analyze the group delay of each of the filters in your model. This section does not show the results for each filter in the `ex_wavelets` model because all wavelet filters in this particular example have the same group delay.

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 5-12
- “Dyadic Synthesis Filter Banks” on page 5-16

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.



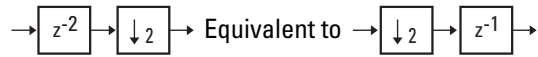
<p>F0 = Delay due to low-pass filter of Dyadic Analysis Filter Bank F1 = Delay due to high-pass filter of Dyadic Analysis Filter Bank G0 = Delay due to low-pass filter of Dyadic Synthesis Filter Bank G1 = Delay due to high-pass filter of Dyadic Synthesis Filter Bank</p>

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.

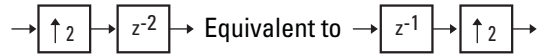
Equalize the Delay on Each Filter Path

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



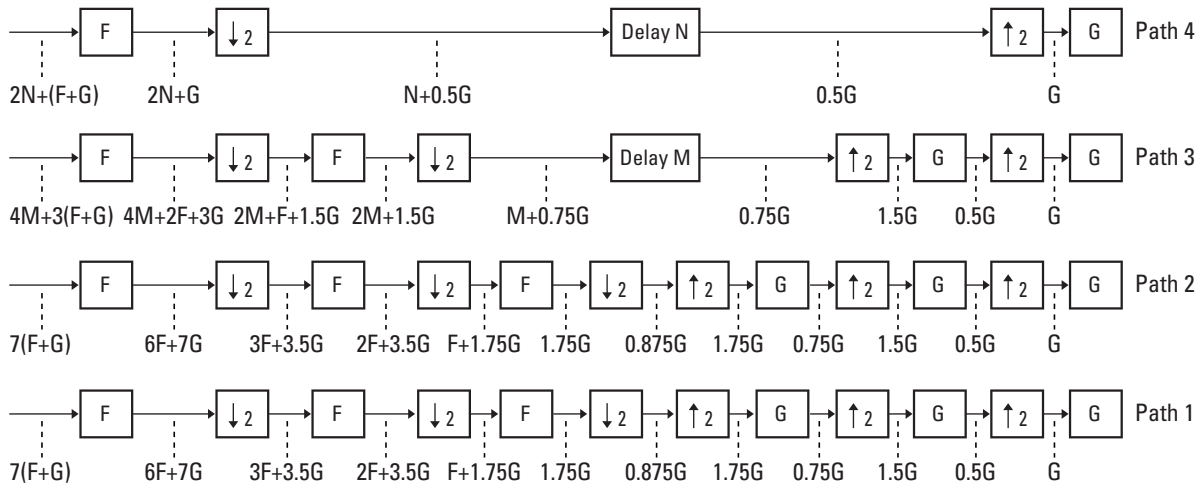
Second Noble Identity



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 downsampling 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 upsampling 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 = 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:

$$\begin{aligned} \text{Path 3} = \text{Path 1} &\Rightarrow 4M + 3(F + G) = 7(F + G) \\ &\Rightarrow M = F + G \end{aligned}$$

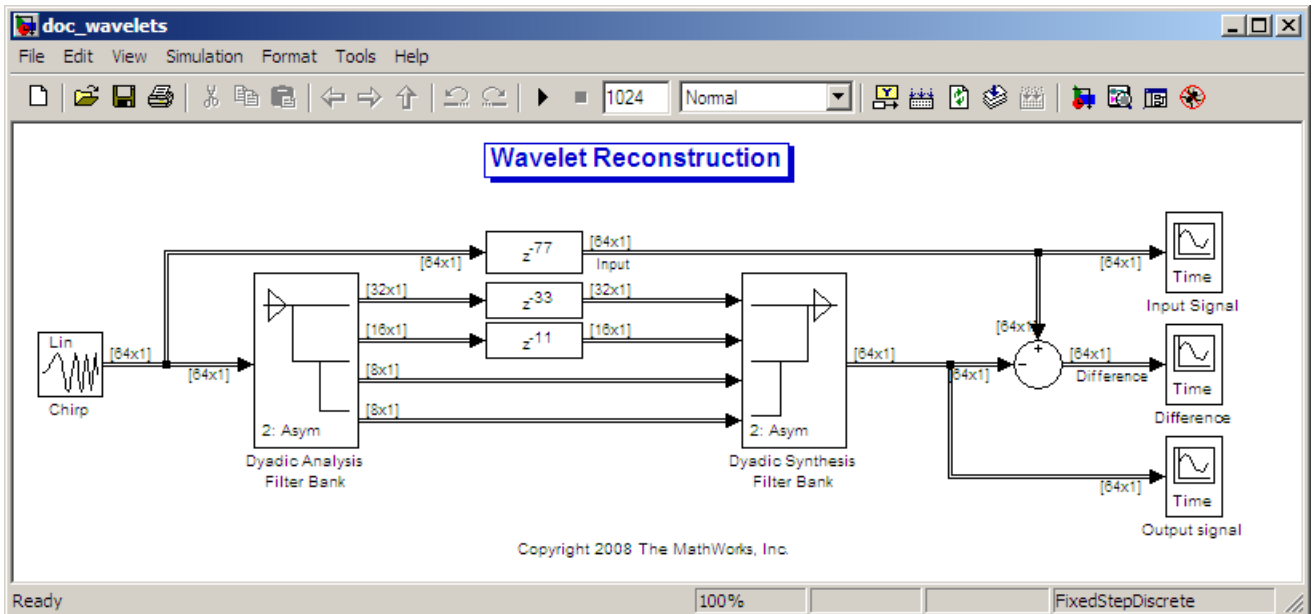
$$\begin{aligned} \text{Path 4} = \text{Path 1} &\Rightarrow 2N + (F + G) = 7(F + G) \\ &\Rightarrow N = 3(F + G) \end{aligned}$$

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 $F = 5.5$

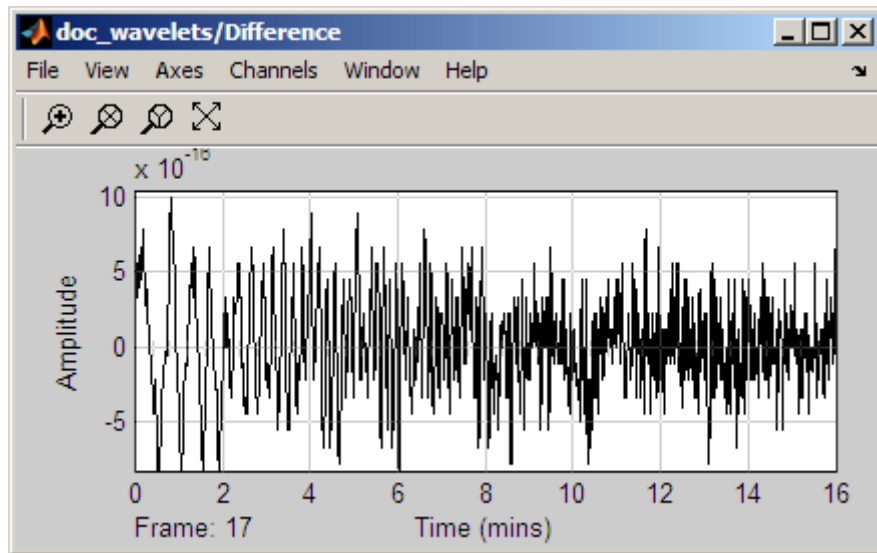
and $G = 5.5$ into the delay equation for any of the four filter paths. Inserting the values of F and G into $7(F+G)$ yields an overall delay of 77 samples for the filter bank system of the `ex_wavelets` model.

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.

Spectral Analysis

The Power Spectrum Estimation library provides a number of blocks for spectral analysis. Many of them have correlates in Signal Processing Toolbox software, which are shown in parentheses:

- Burg Method (`pburg`)
- Covariance Method (`pcov`)
- Magnitude FFT (`periodogram`)
- Modified Covariance Method (`pmcov`)
- Short-Time FFT
- Yule-Walker Method (`pyulear`)

See “Spectral Analysis” in the Signal Processing Toolbox documentation for an overview of spectral analysis theory and a discussion of the above methods.

DSP System Toolbox software provides two demos that illustrate the spectral analysis blocks:

- A Comparison of Spectral Analysis Techniques (`dspsacomp`)
- Spectral Analysis: Short-Time FFT (`dspstfft`)

Power Spectrum Estimates

In this section...

“Create the Block Diagram” on page 6-24

“Set the Model Parameters” on page 6-25

“View the Power Spectrum Estimates” on page 6-31

Create the Block Diagram

Up until now, you have been dealing with signals in the time domain. The DSP System Toolbox product is also capable of working with signals in the frequency domain. You can use the software to perform fast Fourier transforms (FFTs), power spectrum analysis, short-time FFTs, and many other frequency-domain applications.

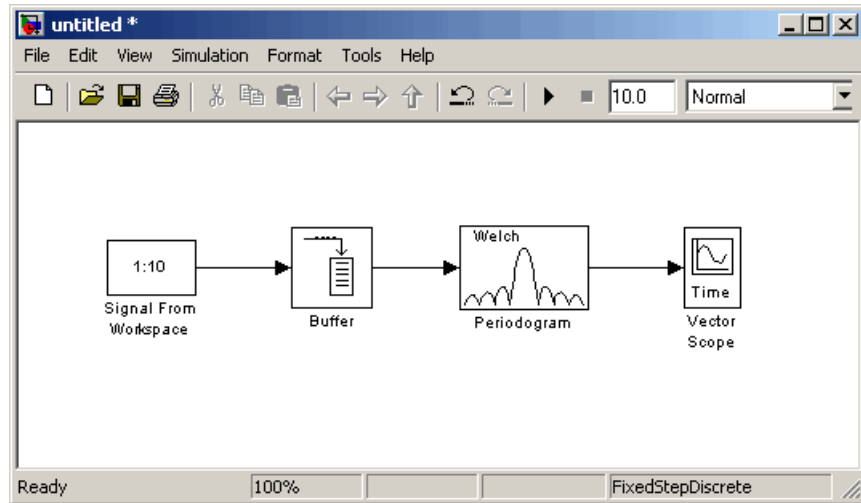
The power spectrum of a signal represents the contribution of every frequency of the spectrum to the power of the overall signal. It is useful because many signal processing applications, such as noise cancellation and system identification, are based on frequency-specific modifications of signals.

First, assemble and connect the blocks needed to calculate the power spectrum of your speech signal:

- 1 Open a new Simulink model.
- 2 Add the following blocks to your model. Subsequent topics describe how to use these blocks.

Block	Library
Signal From Workspace	Sources
Buffer	Signal Management / Buffers
Periodogram	Estimation / Power Spectrum Estimation
Vector Scope	Sinks

- 3 Connect the blocks as shown in the next figure.



Once you have assembled the blocks needed to calculate the power spectrum of your speech signal, you can set the block parameters.

Set the Model Parameters

Now that you have assembled the blocks needed to calculate the power spectrum of your speech signal, you need to set the block parameters. These parameter values ensure that the model calculates the power spectrum of your signal accurately:

- 1 If the model you created in “Create the Block Diagram” on page 6-24 is not open on your desktop, you can open an equivalent model by typing

```
ex_gstut9
```

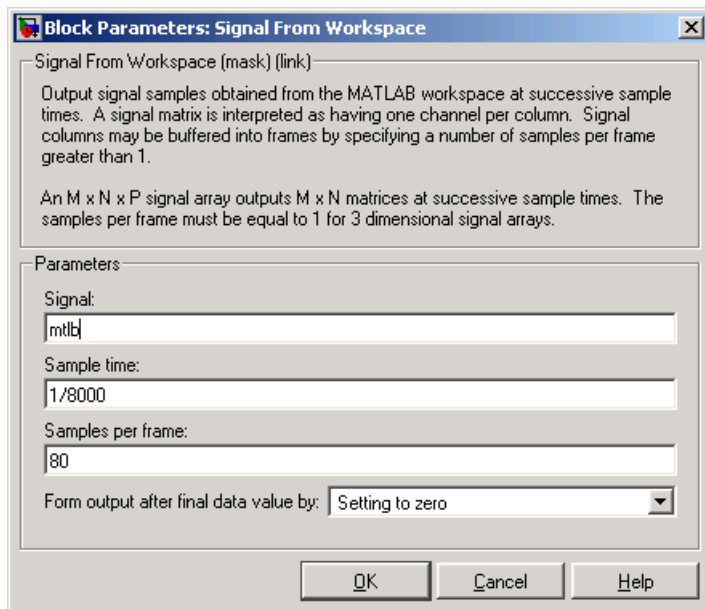
at the MATLAB command prompt.

- 2 Load the speech signal into the MATLAB workspace by typing `load mtlb` at the MATLAB command prompt. This speech signal is a woman’s voice saying “MATLAB.”
- 3 Use the Signal From Workspace block to import the speech signal from the MATLAB workspace into your Simulink model. Open the Signal

From Workspace dialog box by double-clicking the block. Set the block parameters as follows:

- **Signal** = `mtlb`
- **Sample time** = `1/8000`
- **Samples per frame** = `80`
- **Form output after final data value by** = Setting to zero

Once you are done setting these parameters, the Signal From Workspace dialog box should look similar to the figure below. Click **OK** to apply your changes.



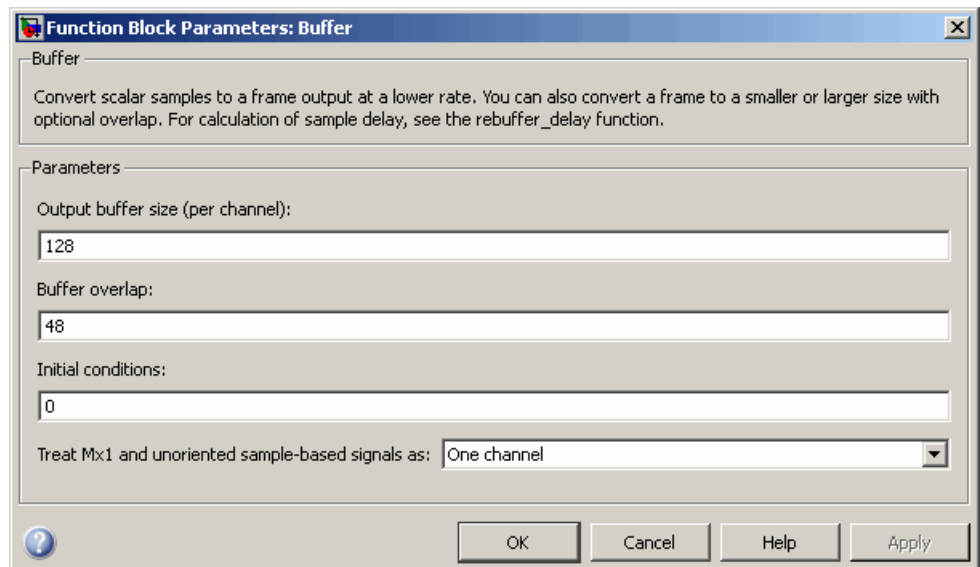
The DSP System Toolbox product is capable of frame-based processing. In other words, DSP System Toolbox blocks can process multiple samples of data at one time. This improves the computational speed of your model. In this case, by setting the **Samples per frame** parameter to **80**, you are telling the Signal From Workspace block to output a frame that contains 80 signal samples at each simulation time step. Note that the sample period

of the input signal is 1/8000 seconds. Also, after the block outputs the final signal value, all other outputs are zero.

- 4 Use the Buffer block to buffer the input signal into frames that contain 128 samples. Open the Buffer dialog box by double-clicking the block. Set the block parameters as follows:

- **Output buffer size (per channel) = 128**
- **Buffer overlap = 48**
- **Initial conditions = 0**
- **Treat Mx1 and unoriented sample-based signals as = One channel**

Once you are done setting these parameters, the Buffer dialog box should look similar to the figure below. Click **OK** to apply your changes.



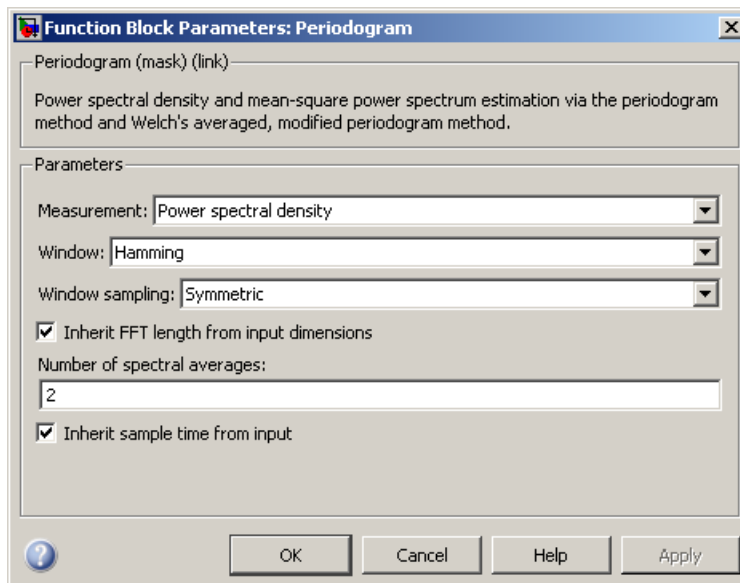
Based on these parameters, the first output frame contains 48 initial condition values followed by the first 80 samples from the first input frame. The second output frame contains the last 48 values from the previous frame followed by the second 80 samples from the second input frame, and so on. You are buffering your input signal into an output signal with 128

samples per frame to minimize the estimation noise added to your signal. Because 128 is a power of 2, this operation also enables the Periodogram block to perform an FFT on the signal.

5 Use the Periodogram block to compute a nonparametric estimate of the power spectrum of the speech signal. Open the Periodogram dialog box by double-clicking the block and set the block parameters as follows:

- **Measurement** = Power spectral density
- **Window** = Hamming
- **Window sampling** = Periodic
- Select the **Inherit FFT length from input dimensions** check box.
- **Number of spectral averages** = 2

Once you are done setting these parameters, the Periodogram dialog box should look similar to the figure below. Click **OK** to apply your changes.

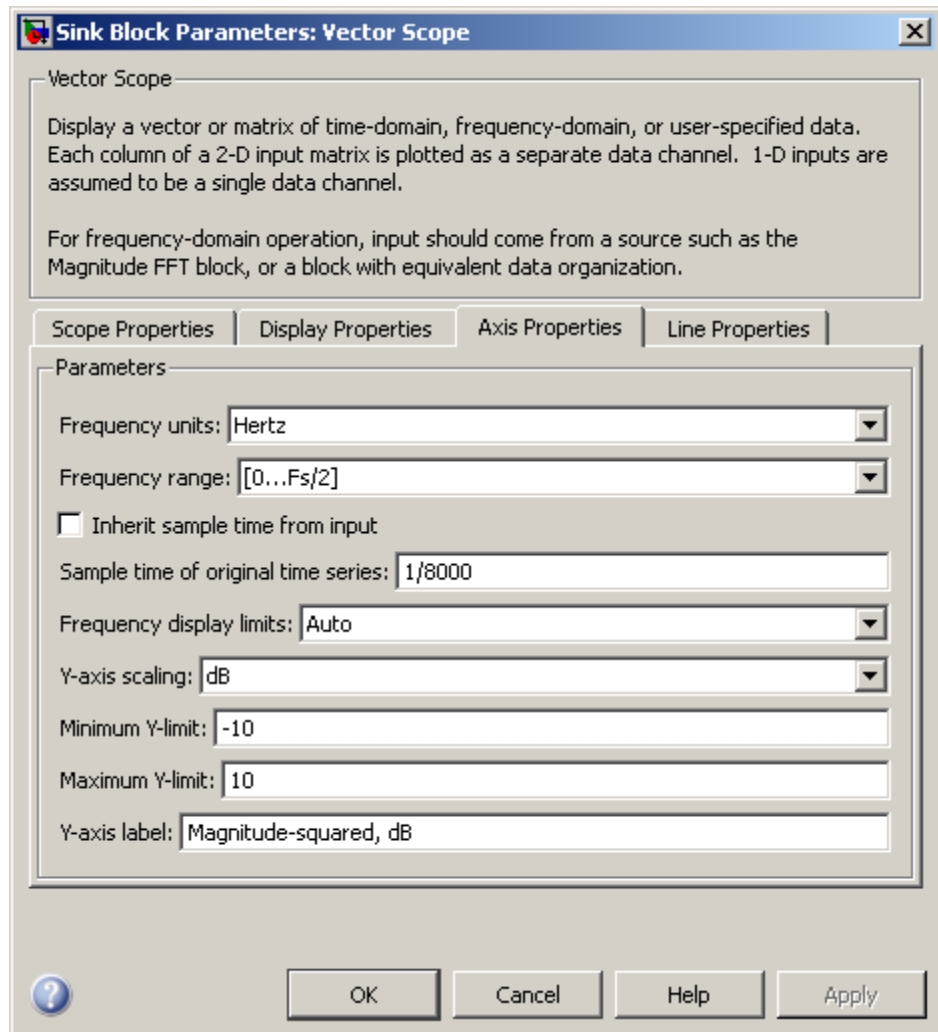


Based on these parameters, the block applies a Hamming window periodically to the input speech signal and averages two spectra at one

time. The length of the FFT is assumed to be 128, which is the number of samples per frame being output from the Buffer block.

- 6 Use the Vector Scope block to view the power spectrum of the speech signal. Open the Vector Scope dialog box by double-clicking the block. Set the block parameters as follows:
 - **Input domain** = Frequency
 - Click the **Axis Properties** tab.
 - Clear the **Inherit sample time from input** check box.
 - **Sample time of original time series** = 1/8000
 - **Y-axis label** = Magnitude-squared, dB

Once you are done setting these parameters, the **Axis Properties** pane of the Vector Scope dialog box should look similar to the figure below. As you can see by the **Y-axis scaling** parameter, the decibel amplitude is plotted in a vector scope window.



Because you are buffering the input with a nonzero overlap, you have altered the sample time of the signal. As a result, you need to specify the sample time of the original time series. Otherwise, the overlapping buffer samples lead the block to believe that the sample time is shorter than it actually is.

After you have set the block parameter values, you can calculate and view the power spectrum of the speech signal.

View the Power Spectrum Estimates

In the previous topics, you created a power spectrum model and set its parameters. In this topic, you simulate the model and view the power spectrum of your speech signal:

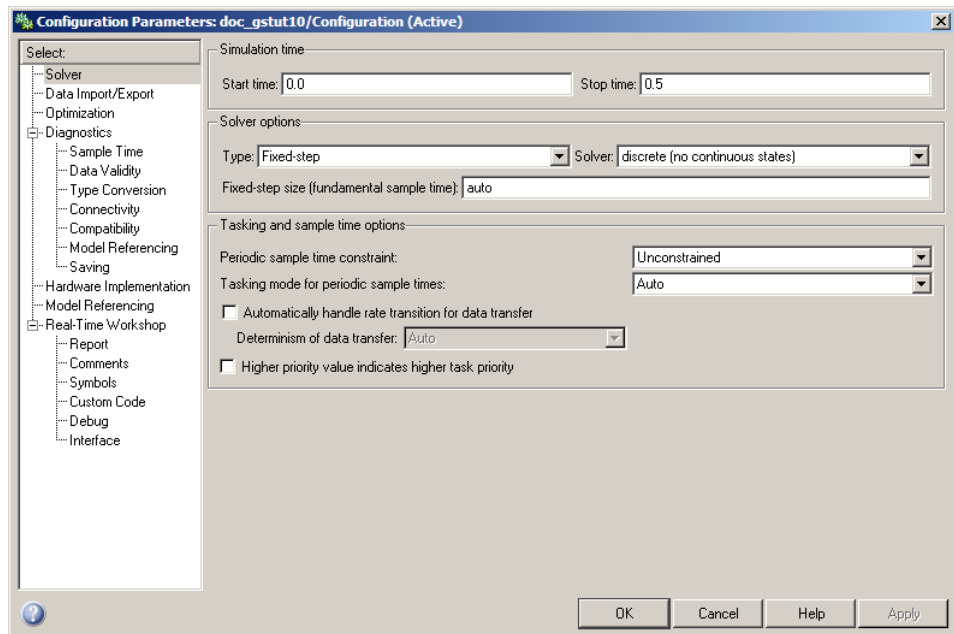
- 1 If the model you created in “Set the Model Parameters” on page 6-25 is not open on your desktop, you can open an equivalent model by typing

```
ex_gstut10
```

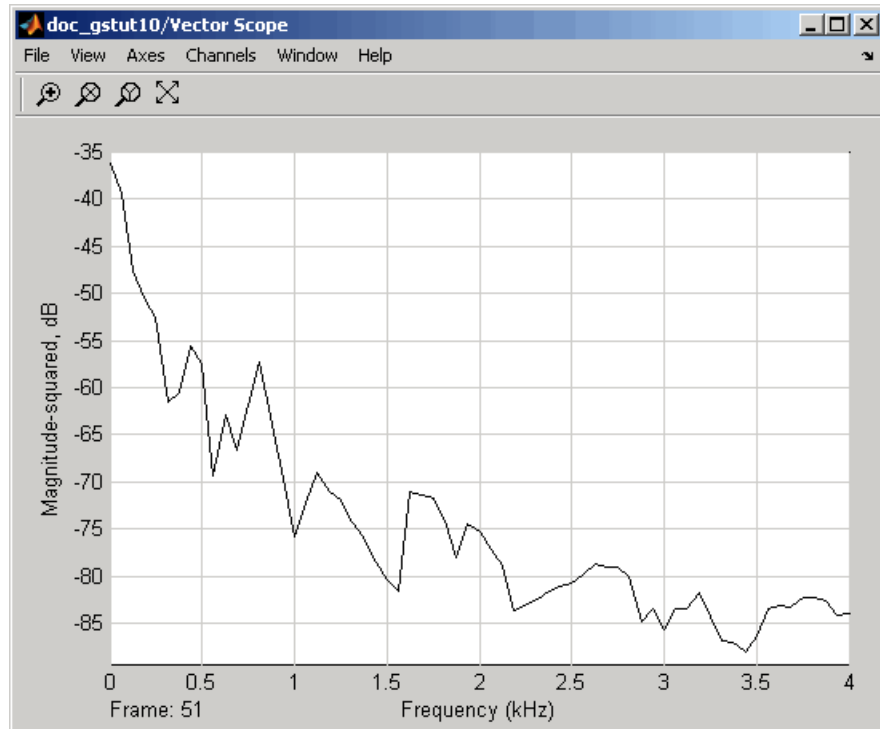
at the MATLAB command prompt.

- 2 Set the configuration parameters. Open the Configuration Parameters dialog box by selecting **Configuration Parameters** from the **Simulation** menu. Select **Solver** from the menu on the left side of the dialog box, and set the parameters as follows:

- **Stop time** = 0.5
- **Type** = Fixed-step
- **Solver** = Discrete (no continuous states)



- 3 Apply these parameters and close the Configuration Parameters dialog box by clicking **OK**. These parameters are saved only when you save your model.
- 4 If you have not already done so, load the speech signal into the MATLAB workspace by typing `load mt1b`.
- 5 Run the model to open the Vector Scope window. The data is not immediately visible at the end of the simulation. To autoscale the *y*-axis to fit the data, in the Vector Scope window, right-click and choose **Autoscale**. The following figure shows the data displayed in the Vector Scope window.



During the simulation, the Vector Scope window displays a series of frames output from the Periodogram block. Each of these frames corresponds to a window of the original speech signal. The data in each frame represents the power spectrum, or contribution of every frequency to the power of the original speech signal, for a given window.

In the next section, “Spectrograms” on page 6-34, you use these power spectra to create a spectrogram of the speech signal.

Spectrograms

In this section...
“Modify the Block Diagram” on page 6-34
“Set the Model Parameters” on page 6-36
“View the Spectrogram of the Speech Signal” on page 6-40

Modify the Block Diagram

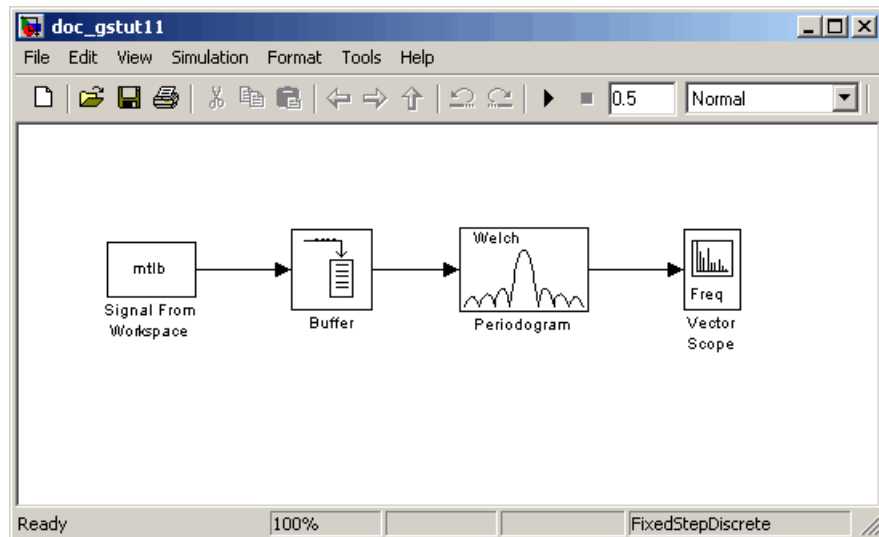
Spectrograms are color-based visualizations of the evolution of the power spectrum of a speech signal as this signal is swept through time. Spectrograms use the periodogram power spectrum estimation method and are widely used by speech and audio engineers. You can use them to develop a visual understanding of the frequency content of your speech signal while a particular sound is being vocalized.

In the previous section, you built a model capable of calculating the power spectrum of a speech signal that represents a woman saying “MATLAB.” In this topic, you modify this model to view the spectrogram of your signal:

- 1 If the model you created in “View the Power Spectrum Estimates” on page 6-31 is not open on your desktop, you can open an equivalent model by typing

```
ex_gstut11
```

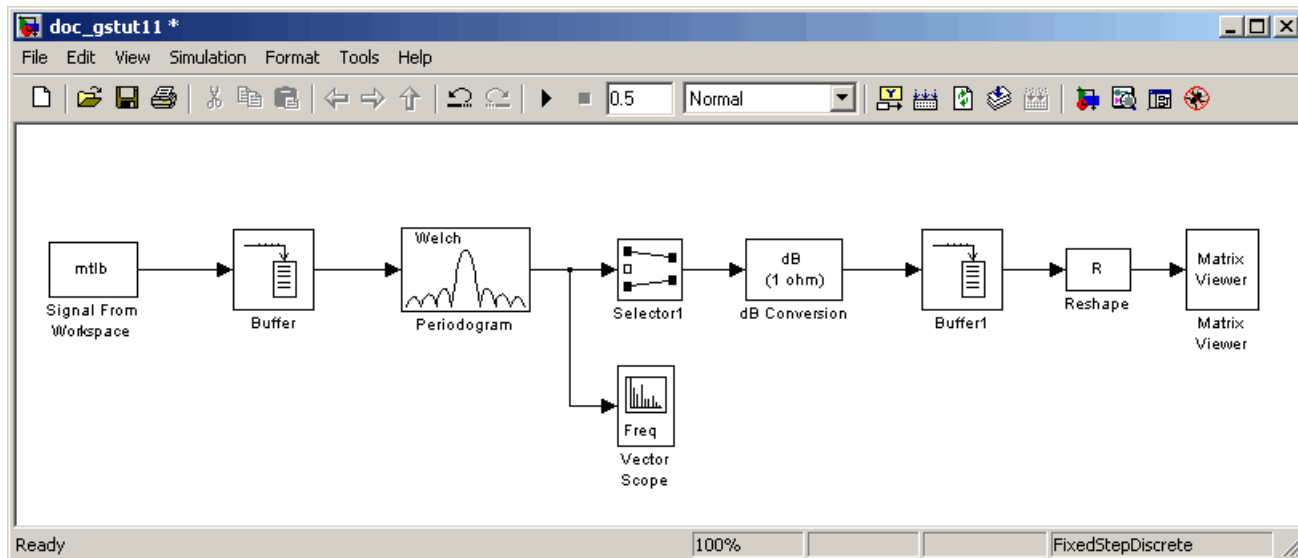
at the MATLAB command prompt.



- 2 Add the following blocks to your model. Subsequent topics describe how to use these blocks.

Block	Library
Selector	Simulink / Signal Routing
dB Conversion	Math Functions / Math Operations
Buffer	Signal Management / Buffers
Reshape	Simulink / Math Operations
Matrix Viewer	Sinks

- 3 Connect the blocks as shown in the figure below. These blocks extract the positive frequencies of each power spectrum and concatenate them into a matrix that represents the spectrogram of the speech signal.



Once you have assembled the blocks needed to view the spectrogram of your speech signal, you can set the block parameters.

Set the Model Parameters

In the previous topic, you assembled the blocks you need to view the spectrogram of your speech signal. Now you must set the block parameters:

- 1 If the model you created in “Modify the Block Diagram” on page 6-34 is not open on your desktop, you can open an equivalent model by typing

`ex_gstut12`

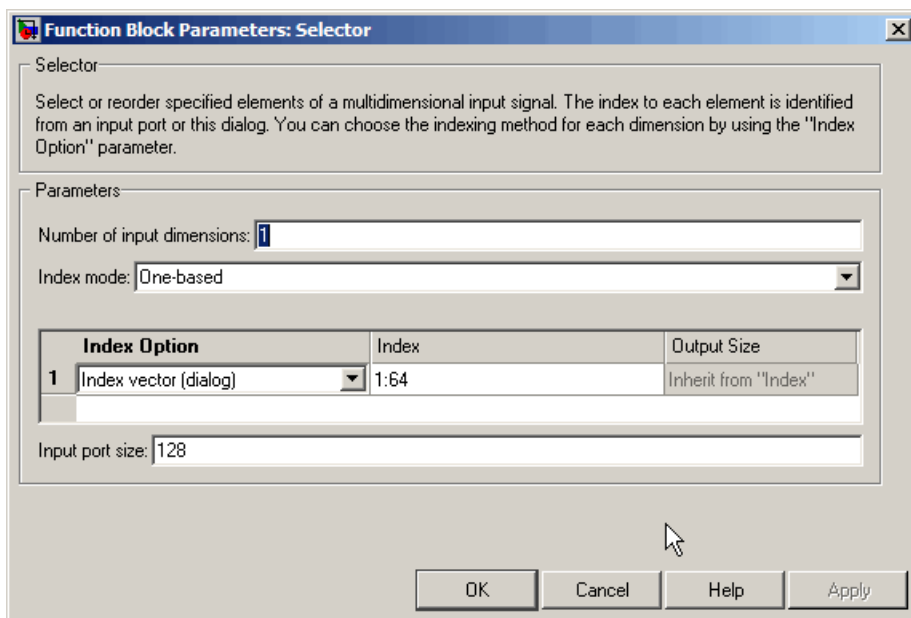
at the MATLAB command prompt.

- 2 Use the Selector block to extract the first 64 elements, or the positive frequencies, of each power spectrum. Open the Selector dialog box by double-clicking the block. Set the block parameters as follows:

- **Number of input dimensions** = 1
- **Index mode** = One-based

- **Index option** = Index vector (dialog)
- **Index** = 1:64
- **Input port size** = 128

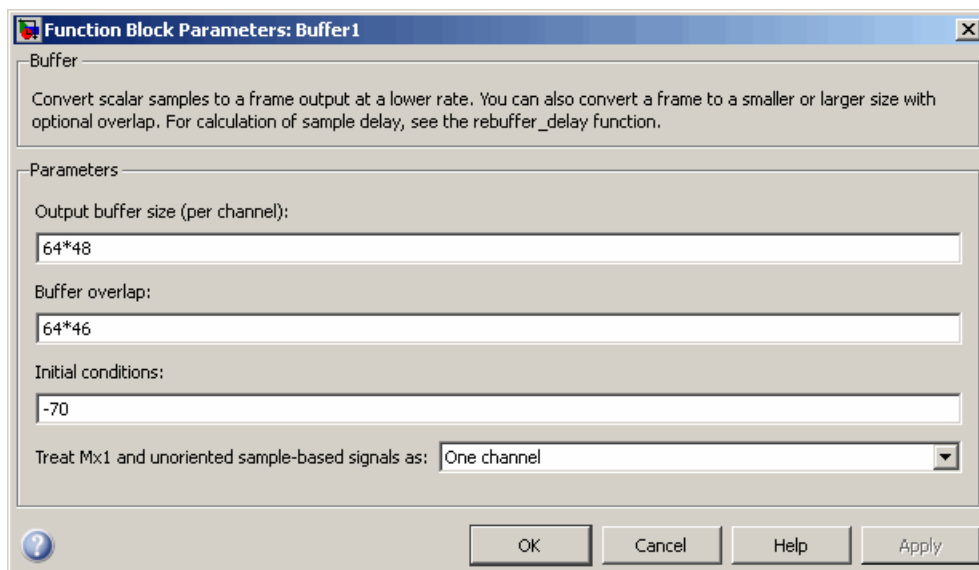
At each time instance, the input to the Selector block is a vector of 128 elements. The block assigns one-based indices to these elements and extracts the first 64. Once you are done setting these parameters, the Selector dialog box should look similar to the figure below. To apply your changes, click **OK**.



- 3 The dB Conversion block converts the magnitude of the input FFT signal to decibels. Leave this block at its default parameters.
- 4 Use the Buffer1 block to buffer up the individual power spectrums. Open the Buffer1 dialog box by double-clicking the block. Set the block parameters as follows:
 - **Output buffer size (per channel)** = 64×48
 - **Buffer overlap** = 64×46

- **Initial conditions** = -70
- **Treat Mx1 and unoriented sample-based signals as** = One channel

Once you are done setting these parameters, the Buffer1 dialog box should look similar to the following figure. To apply your changes, click **OK**.

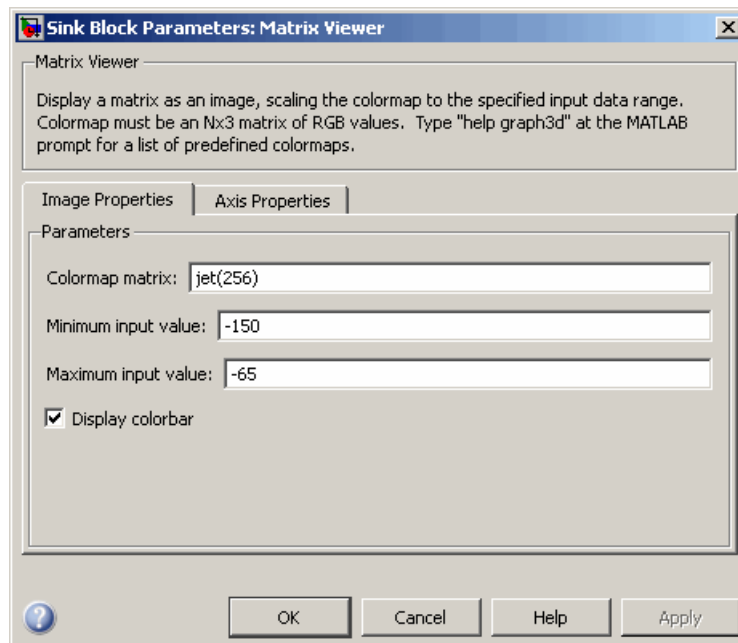


Setting the value of the **Buffer overlap** parameter slightly less than the value of the **Output buffer size (per channel)** parameter ensures that your spectrogram represents smooth movement through time. The **Initial conditions** parameter represents the initial values in the buffer; -70 represents silence.

- 5 Use the Reshape block to reshape the input signal into a 64-by-48 matrix. To do so, set the **Output dimensionality** to **Customize** and the **Output dimensions** to [64 48].
- 6 The Matrix Viewer enables you to view the spectrogram of the speech signal. Open the Matrix Viewer dialog box by double-clicking the block. Set the block parameters as follows:
 - Click the **Image Properties** tab.

- **Colormap matrix** = `jet(256)`
- **Minimum input value** = -150
- **Maximum input value** = -65
- Select the **Display colorbar** check box.

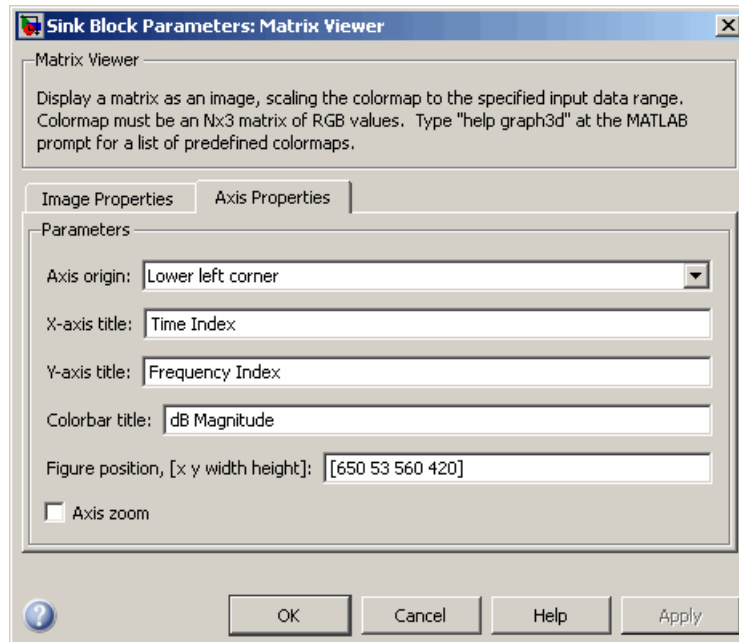
Once you are done setting these parameters, the **Image Properties** pane should look similar to the figure below.



- Click the **Axis Properties** tab.
- **Axis origin** = Lower left corner
- **X-axis title** = Time Index
- **Y-axis title** = Frequency Index
- **Colorbar title** = dB Magnitude

In this case, you are assuming that the power spectrum values do not exceed -65 dB. Once you are done setting these parameters, the **Axis**

Properties pane should look similar to the figure below. To apply your changes, click **OK**.



After you have set the parameter values, you can calculate and view the spectrogram of the speech signal.

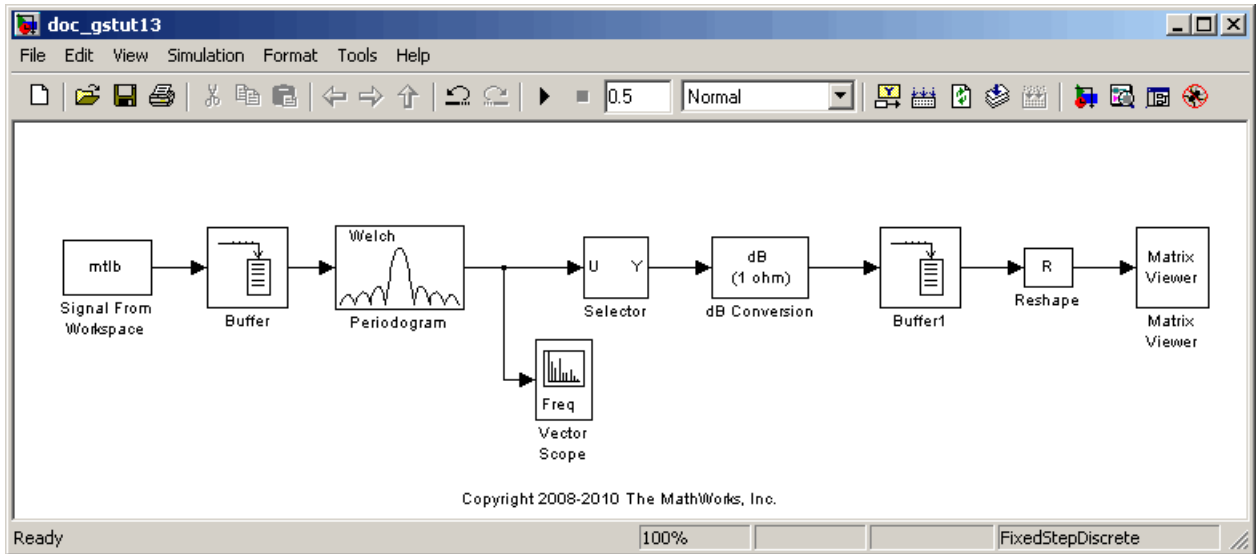
View the Spectrogram of the Speech Signal

In the topic “View the Power Spectrum Estimates” on page 6-31, you used a Vector Scope block to display the power spectrum of your speech signal. In this topic, you view the spectrogram of your speech signal using a Matrix Viewer block. The speech signal represents a woman’s voice saying “MATLAB”:

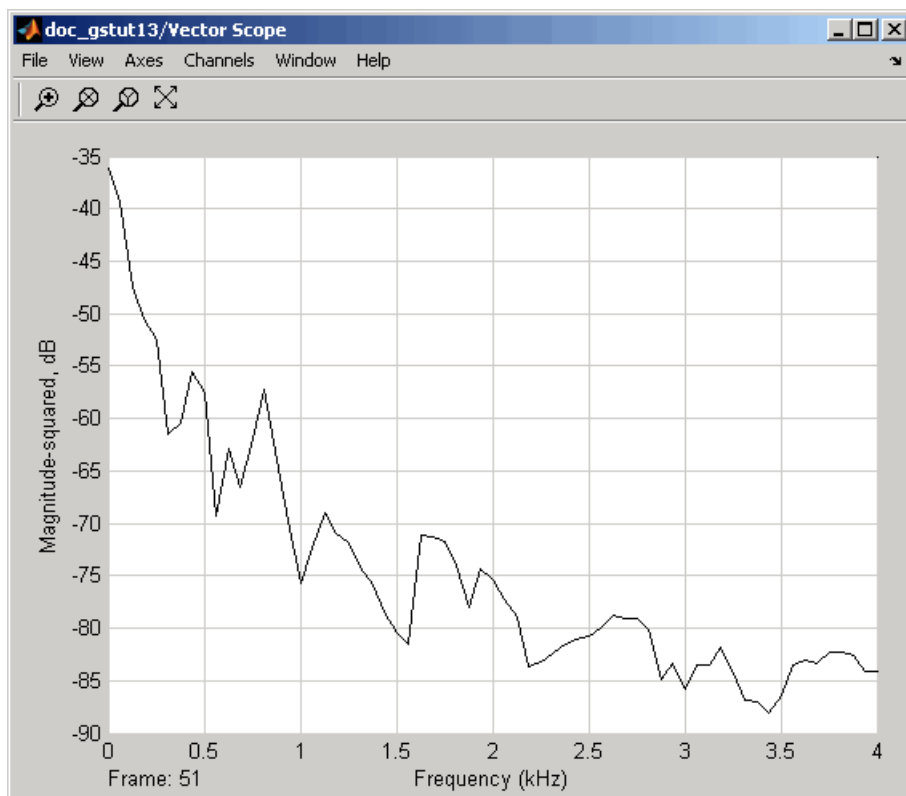
- 1 If the model you created in “Set the Model Parameters” on page 6-36 is not open on your desktop, you can open an equivalent model by typing

```
ex_gstut13
```

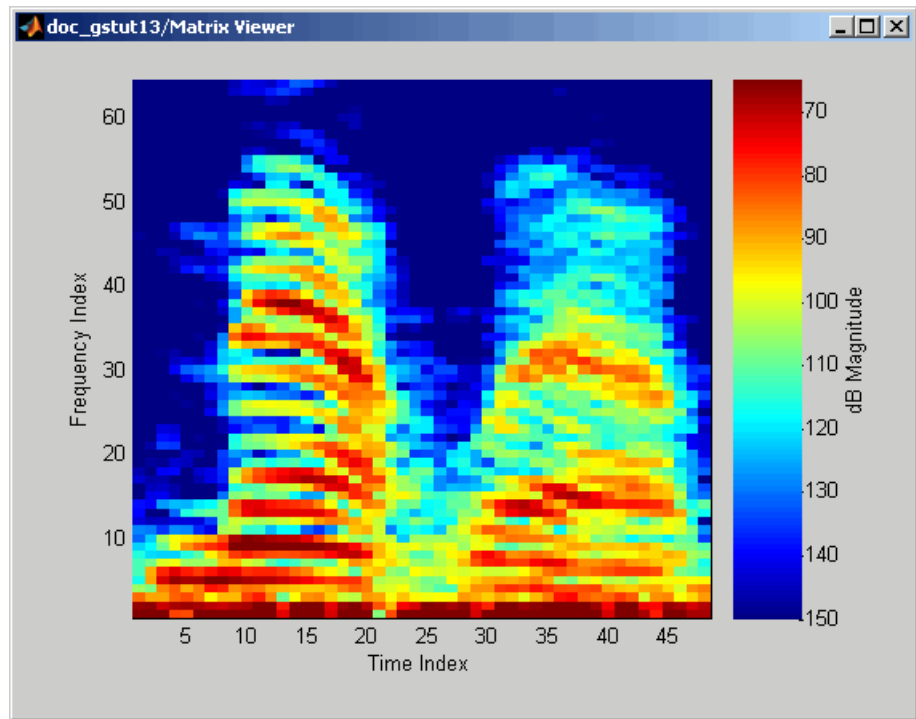
at the MATLAB command prompt.



- 2 Run the model. During the simulation, the Vector Scope window displays a sequence of power spectrums, one for each window of the original speech signal. The power spectrum is the contribution of every frequency to the power of the speech signal.



The Matrix Viewer window, shown below, displays the spectrogram of the speech signal. This spectrogram is calculated using the Periodogram power spectrum estimation method. Note the harmonics that are visible in the signal when the vowels are spoken. Most of the signal's energy is concentrated in these harmonics; therefore, two distinct peaks are visible in the spectrogram.



In this example, you viewed the spectrogram of your speech signal using a Matrix Viewer block. You can find additional DSP System Toolbox product examples in the Help browser. To access these examples, click the **Contents** tab, double-click **DSP System Toolbox**, and then click **Examples**. A list of the examples in the DSP System Toolbox documentation appears in the right pane of the Help browser.

Mathematics

Learn about statistics and linear algebra.

- “Statistics” on page 7-2
- “Linear Algebra” on page 7-6

Statistics

In this section...
“Statistics Blocks” on page 7-2
“Basic Operations” on page 7-3
“Running Operations” on page 7-4

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
```

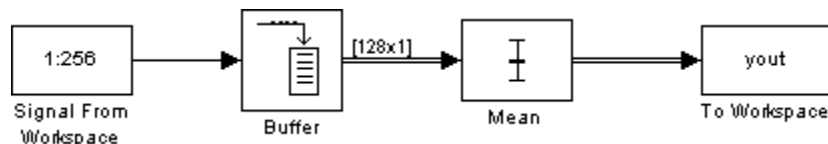
which computes the maximum of each column in input *u*.

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.



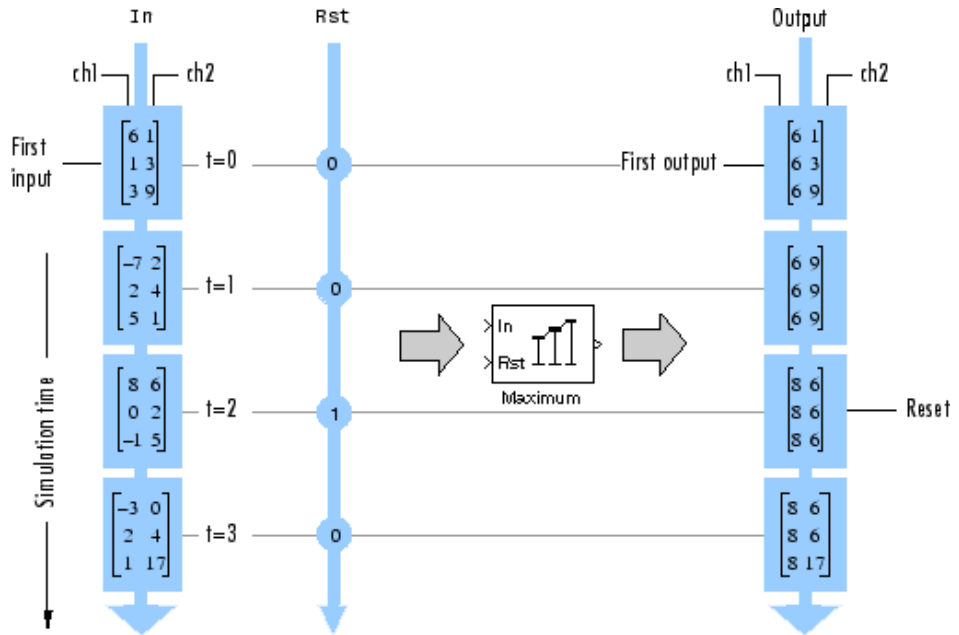
The **Buffer** block's **Buffer size** (M_o) parameter determines the size of the window. The **Buffer overlap** (L) parameter defines the “slide factor” for the window. At each sample instant, the window slides by $M_o - L$ points. The **Buffer overlap** is often $M_o - 1$, so that a new statistic is 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

In this section...
“Linear Algebra Blocks” on page 7-6
“Linear System Solvers” on page 7-6
“Matrix Factorizations” on page 7-8
“Matrix Inverses” on page 7-9

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

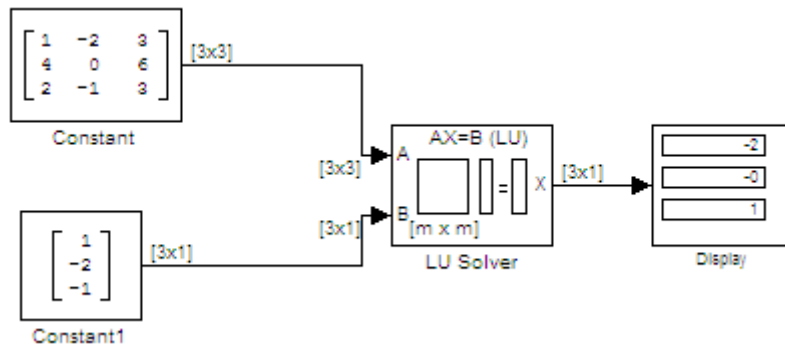
Some of the blocks offer particular strengths for certain classes of problems. For example, the Cholesky Solver block is particularly adapted for a square Hermitian positive definite matrix A , whereas the Backward Substitution block is particularly suited for an upper triangular matrix A .

Solve $AX=B$ Using the LU Solver Block

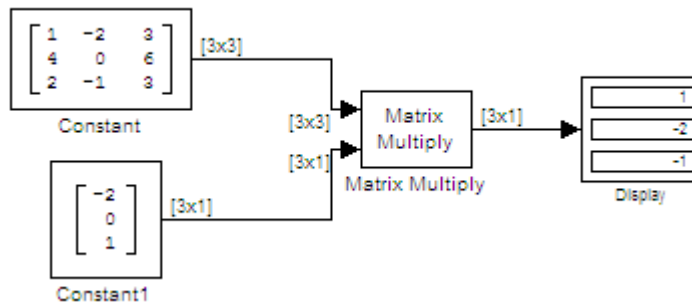
In the following `ex_lusolver_tut` model, the LU Solver block solves the equation $Ax = b$, where

$$A = \begin{bmatrix} 1 & -2 & 3 \\ 4 & 0 & 6 \\ 2 & -1 & 3 \end{bmatrix} \quad b = \begin{bmatrix} 1 \\ -2 \\ -1 \end{bmatrix}$$

and finds x to be the vector $[-2 \ 0 \ 1]^T$.



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:

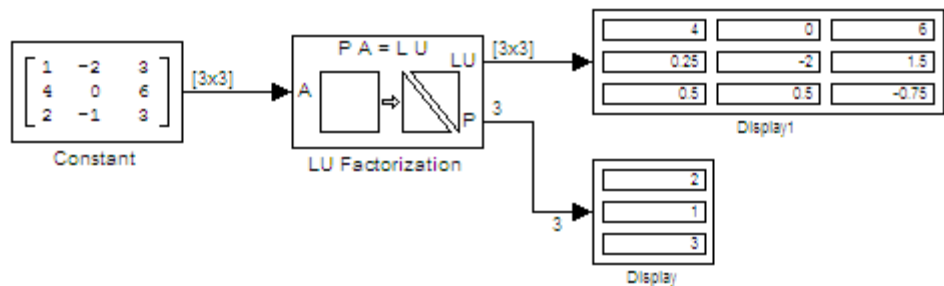
- Cholesky Factorization
- LDL Factorization
- LU Factorization
- QR Factorization
- Singular Value Decomposition

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_lufacturation_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

$$A = \begin{bmatrix} 1 & -2 & 3 \\ 4 & 0 & 6 \\ 2 & -1 & 3 \end{bmatrix}$$



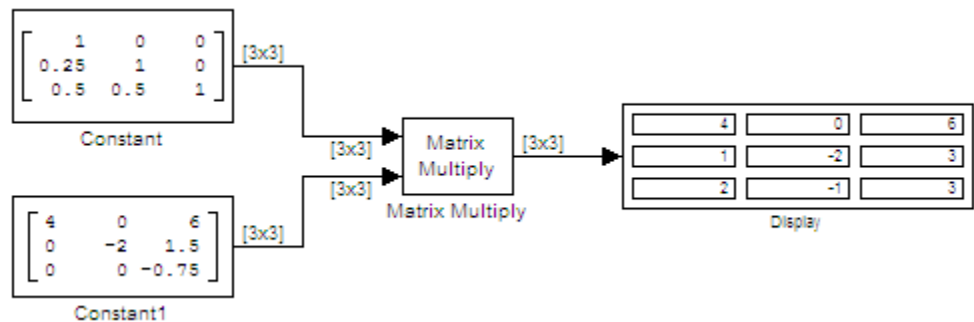
The lower output of the LU Factorization, P , is the permutation index vector, which indicates that the factored matrix A_p is generated from A by interchanging the first and second rows.

$$\mathbf{A}_p = \begin{bmatrix} 4 & 0 & 6 \\ 1 & -2 & 3 \\ 2 & -1 & 3 \end{bmatrix}$$

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

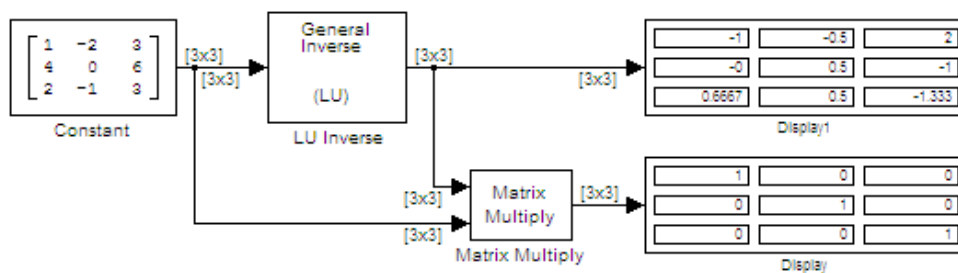
- LU Inverse
- Pseudoinverse

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 $\mathbf{A}^{-1}\mathbf{A}$, which yields the identity matrix of order 3, as expected.



As shown above, the computed inverse is

$$\mathbf{A}^{-1} = \begin{bmatrix} -1 & -0.5 & 2 \\ 0 & 0.5 & -1 \\ 0.6667 & 0.5 & -1.333 \end{bmatrix}$$

Fixed-Point Design

Learn about fixed-point data types and how to convert floating-point models to fixed-point.

- “Fixed-Point Signal Processing” on page 8-2
- “Fixed-Point Concepts and Terminology” on page 8-4
- “Arithmetic Operations” on page 8-10
- “Fixed-Point Support for MATLAB System Objects” on page 8-21
- “Specify Fixed-Point Attributes for Blocks” on page 8-28
- “Quantizers” on page 8-51
- “Fixed-Point Filter Design” on page 8-65

Fixed-Point Signal Processing

In this section...
“Fixed-Point Features” on page 8-2
“Benefits of Fixed-Point Hardware” on page 8-2
“Benefits of Fixed-Point Design with System Toolboxes Software” on page 8-3

Note To take full advantage of fixed-point support in System Toolbox software, you must install Simulink Fixed Point 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. For more information on generating fixed-point code, see Code Generation.

Fixed-Point Concepts and Terminology

In this section...

“Fixed-Point Data Types” on page 8-4

“Scaling” on page 8-5

“Precision and Range” on page 8-6

Note The “Glossary” defines much of the vocabulary used in these sections. For more information on these subjects, see the Simulink Fixed Point and *Fixed-Point Toolbox™ User’s Guide* documentation.

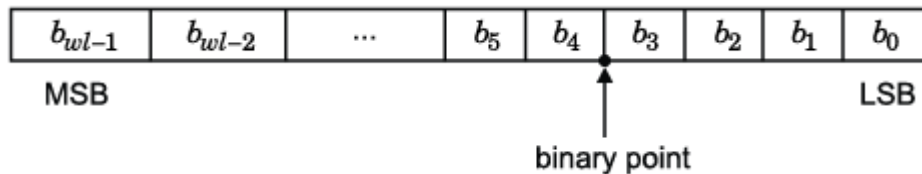
Fixed-Point Data Types

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 i^{th} 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 8-11 for more information.

Scaling

Fixed-point numbers can be encoded according to the scheme

$$\text{real-world value} = (\text{slope} \times \text{integer}) + \text{bias}$$

where the slope can be expressed as

$$\text{slope} = \text{slope adjustment} \times 2^{\text{exponent}}$$

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 Simulink Fixed Point [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:

$$\text{real-world value} = 2^{\text{exponent}} \times \text{integer}$$

or

$$\text{real-world value} = 2^{-\text{fraction length}} \times \text{integer}$$

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.

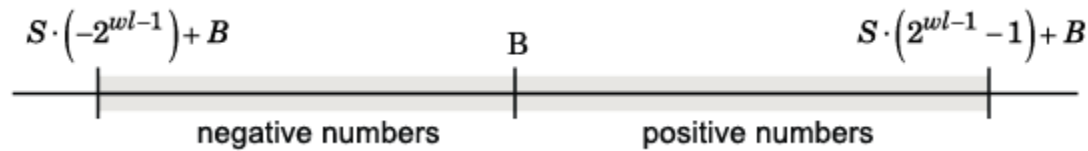
All System Toolbox blocks that support fixed-point data types support signals with binary-point only scaling. Many fixed-point blocks that do not perform arithmetic operations but merely rearrange data, such as Delay and Matrix Transpose, also support signals with [Slope Bias] scaling.

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

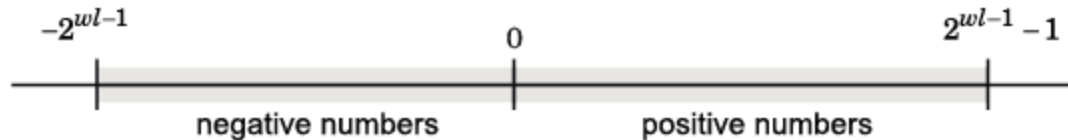
The range is the span of numbers that a fixed-point data type and scaling can represent. The range of representable numbers for a two's complement fixed-point number of word length wl , scaling S , and bias B is illustrated below:



For both signed and unsigned fixed-point numbers of any data type, the number of different bit patterns is $2wl$.

For example, in two's complement, negative numbers must be represented as well as zero, so the maximum value is $2wl^{-1}$. 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 $-2wl^{-1}$ but not for $2wl^{-1}$:

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 8-10 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 Simulink Fixed Point 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 Simulink Fixed Point documentation.

For a direct comparison of the rounding modes, see “Choosing a Rounding Method” in the Fixed-Point Toolbox documentation.

Arithmetic Operations

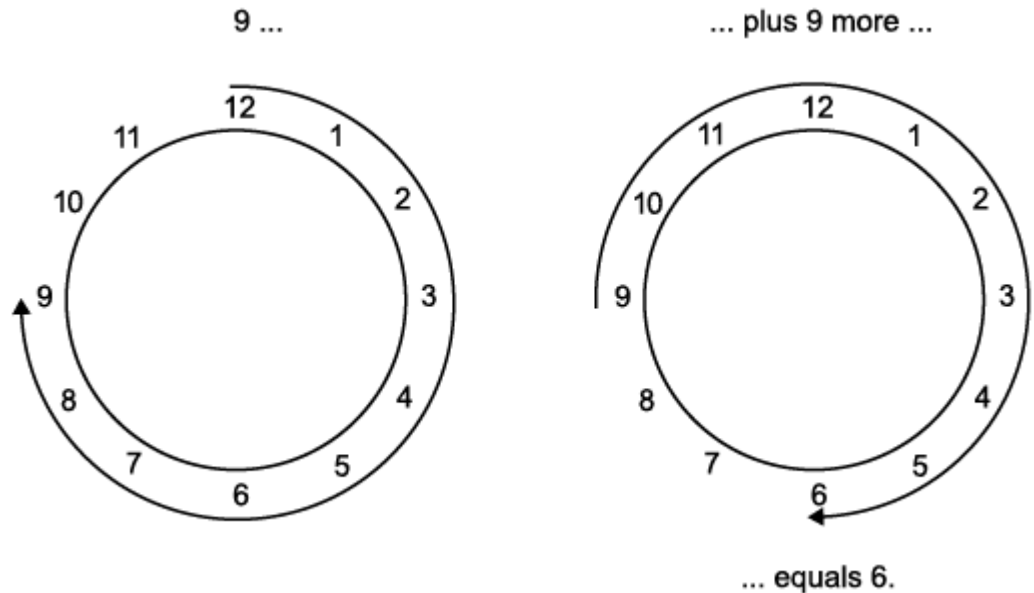
In this section...
“Modulo Arithmetic” on page 8-10
“Two’s Complement” on page 8-11
“Addition and Subtraction” on page 8-12
“Multiplication” on page 8-13
“Casts” on page 8-16

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:



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:

$$11010 \rightarrow 00101$$

Next, add a 1, wrapping all numbers to 0 or 1:

$$\begin{array}{r} 00101 \\ +1 \\ \hline 00110 \text{ (6)} \end{array}$$

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):

$$\begin{array}{r} 010010.1 \quad (18.5) \\ +0110.110 \quad (6.75) \\ \hline 011001.010 \quad (25.25) \end{array}$$

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):

$$\begin{array}{r}
 010010.100 \text{ (18.5)} \\
 - 0110.110 \text{ (6.75)} \\
 \hline
 \end{array}
 \xrightarrow[\text{and sign extension}]{\text{two's complement}}
 \begin{array}{r}
 010010.100 \text{ (18.5)} \\
 +111001.010 \text{ (-6.75)} \\
 \hline
 0001011.110 \text{ (11.75)}
 \end{array}$$

Carry bit is discarded.

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 8-16 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):

$$\begin{array}{r}
 10.11 \text{ (-1.25)} \\
 \quad 011 \text{ (3)} \\
 \hline
 11011 \\
 \quad 1011 \\
 \hline
 1100.01 \text{ (-3.75)}
 \end{array}$$

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

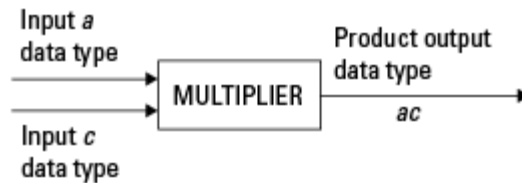
The following diagrams show the data types used for fixed-point multiplication in the System Toolbox software. The diagrams illustrate the differences between the data types used for real-real, complex-real, and complex-complex

multiplication. See individual reference pages to determine whether a particular block accepts complex fixed-point inputs.

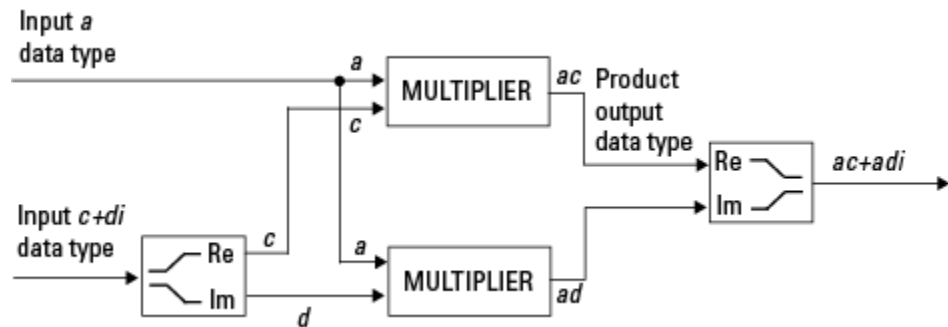
In most cases, you can set the data types used during multiplication in the block mask. See Accumulator Parameters, Intermediate Product Parameters, Product Output Parameters, and Output Parameters. These data types are defined in “Casts” on page 8-16.

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.

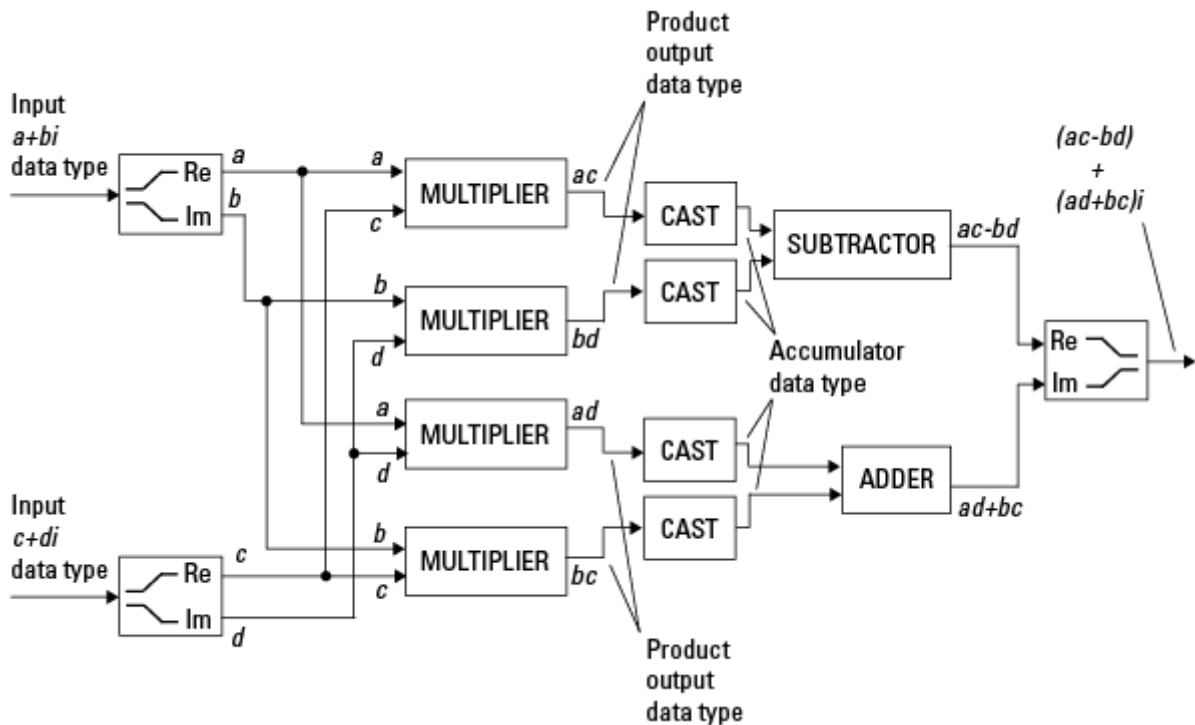
Real-Real Multiplication. The following diagram shows the data types used in the multiplication of two real numbers in System Toolbox software. The software returns the output of this operation in the product output data type, as the next figure shows.



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

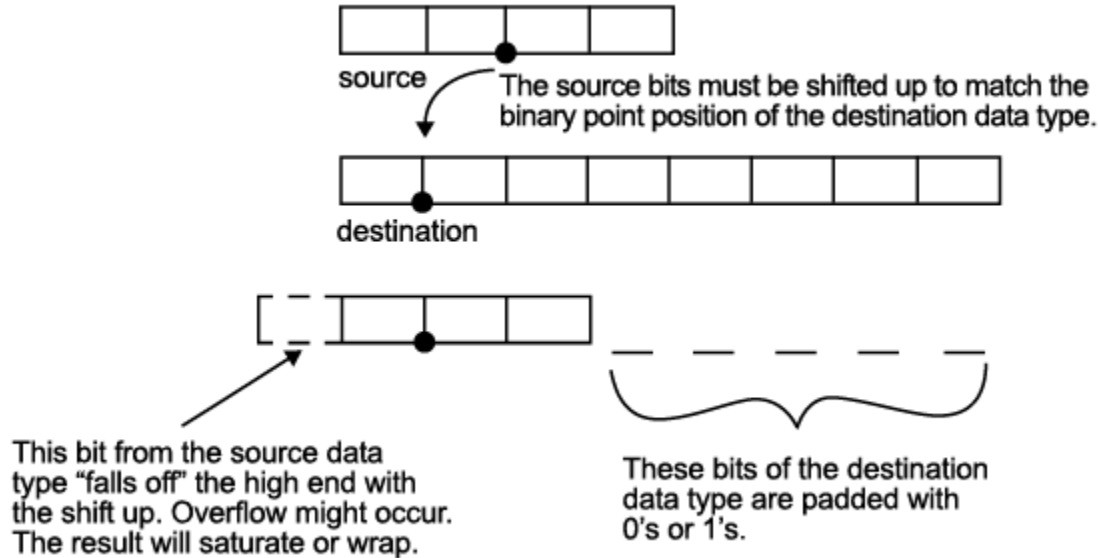
Many fixed-point System Toolbox blocks allow you to specify the data type and scaling of the block output on the mask. Remember that the software does not allow mixed types on the input and output ports of its blocks. Therefore, if you would like to specify a fixed-point output data type and scaling 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:



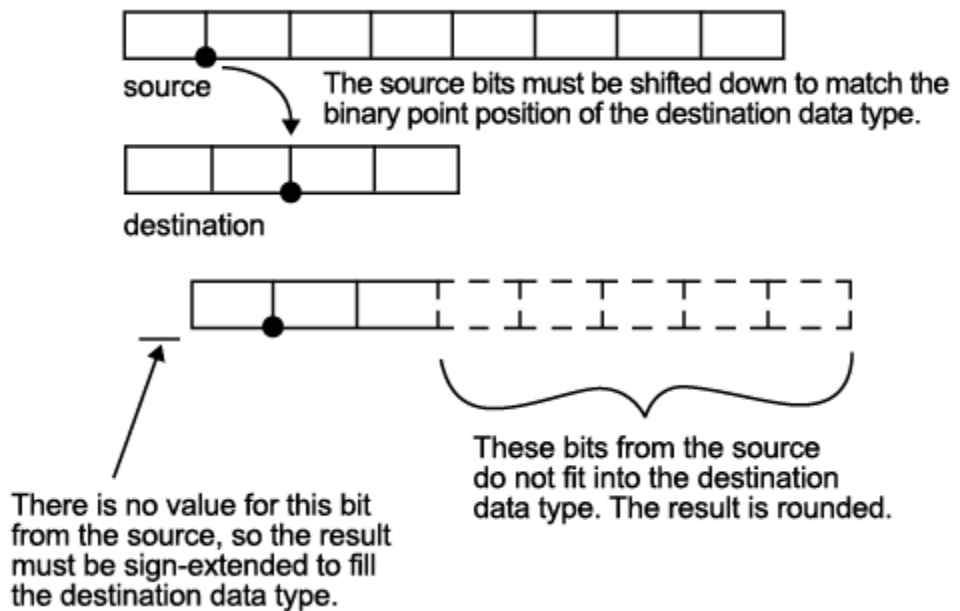
As the diagram shows, the source bits are shifted up so that the binary point matches the destination binary point position. The highest source bit does not fit, so overflow might occur and the result can saturate or wrap. The empty bits at the low end of the destination data type are padded with either 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 this section...

“Get Information About Fixed-Point System Objects” on page 8-21

“Display Fixed-Point Properties for System Objects” on page 8-25

“Set System Object Fixed-Point Properties” on page 8-26

“Full Precision for Fixed-Point System Objects” on page 8-27

Get Information About Fixed-Point System Objects

System objects that support fixed-point data processing have fixed-point properties, which you can display for a particular object by typing `dsp.<ObjectName>.helpFixedPoint` at the command line. See “Display Fixed-Point Properties for System Objects” on page 8-25 to set the display of System object fixed-point properties.

The following signal processing System objects support fixed-point data processing.

DSP System Toolbox System Objects that Support Fixed Point

Object	Description
Estimation	
<code>dsp.LevinsonSolver</code>	Solve linear system of equations using Levinson-Durbin recursion
Filters	
<code>dsp.BiquadFilter</code>	Model biquadratic IIR (SOS) filters
<code>dsp.CICDecimator</code>	Decimate inputs using a Cascaded Integrator-Comb (CIC) filter
<code>dsp.CICInterpolator</code>	Interpolate inputs using a Cascaded Integrator-Comb (CIC) filter
<code>dsp.DigitalFilter</code>	Filter each channel of input over time using discrete-time filter implementations

(Continued)

Object	Description
<code>dsp.FIRDecimator</code>	Filter and downsample input signals
<code>dsp.FIRInterpolator</code>	Upsample and filter input signals
<code>dsp.FIRRateConverter</code>	Upsample, filter and downsample input signals
<code>dsp.LMSFilter</code>	Compute output, error, and weights using LMS adaptive algorithm
<code>dsp.SubbandAnalysisFilter</code>	Decompose signal into high-frequency and low-frequency subbands
<code>dsp.SubbandSynthesisFilter</code>	Reconstruct a signal from high-frequency and low-frequency subbands
Math Functions	
<code>dsp.ArrayVectorAdder</code>	Add vector to array along specified dimension
<code>dsp.ArrayVectorDivider</code>	Divide array by vector along specified dimension
<code>dsp.ArrayVectorMultiplier</code>	Multiply array by vector along specified dimension
<code>dsp.ArrayVectorSubtractor</code>	Subtract vector from array along specified dimension
<code>dsp.CumulativeProduct</code>	Compute cumulative product of channel, column, or row elements
<code>dsp.CumulativeSum</code>	Compute cumulative sum of channel, column, or row elements
<code>dsp.LDLFactor</code>	Factor square Hermitian positive definite matrices into lower, upper, and diagonal components
<code>dsp.LevinsonSolver</code>	Solve linear system of equations using Levinson-Durbin recursion

(Continued)

Object	Description
<code>dsp.LowerTriangularSolver</code>	Solve $LX = B$ for X when L is lower triangular matrix
<code>dsp.LUFactor</code>	Factor square matrix into lower and upper triangular matrices
<code>dsp.Normalizer</code>	Normalize input
<code>dsp.UpperTriangularSolver</code>	Solve $UX = B$ for X when U is upper triangular matrix
Quantizers	
<code>dsp.ScalarQuantizerDecoder</code>	Convert each index value into quantized output value
<code>dsp.ScalarQuantizerEncoder</code>	Perform scalar quantization encoding
<code>dsp.VectorQuantizerDecoder</code>	Find vector quantizer codeword for given index value
<code>dsp.VectorQuantizerEncoder</code>	Perform vector quantization encoding
Signal Management	
<code>dsp.Buffer</code>	Buffer an input signal
<code>dsp.Counter</code>	Count up or down through specified range of numbers
Signal Operations	
<code>dsp.Convolver</code>	Compute convolution of two inputs
<code>dsp.DigitalDownConverter</code>	Digitally down convert the input signal
<code>dsp.DigitalUpConverter</code>	Digitally up convert the input signal
<code>dsp.NCO</code>	Generate real or complex sinusoidal signals
<code>dsp.PeakFinder</code>	Determine extrema (maxima or minima) in input signal

(Continued)

Object	Description
dsp.VariableFractionalDelay	Delay input by time-varying fractional number of sample periods
Sinks	
dsp.SignalLogger	Log MATLAB simulation data
dsp.TimeScope	Display time-domain signals
Sources	
dsp.SignalReader	Import a variable from the MATLAB workspace
dsp.SineWave	Generate discrete sine wave
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
Transforms	
dsp.DCT	Compute discrete cosine transform (DCT) of input
dsp.FFT	Compute fast Fourier transform (FFT) of input

(Continued)

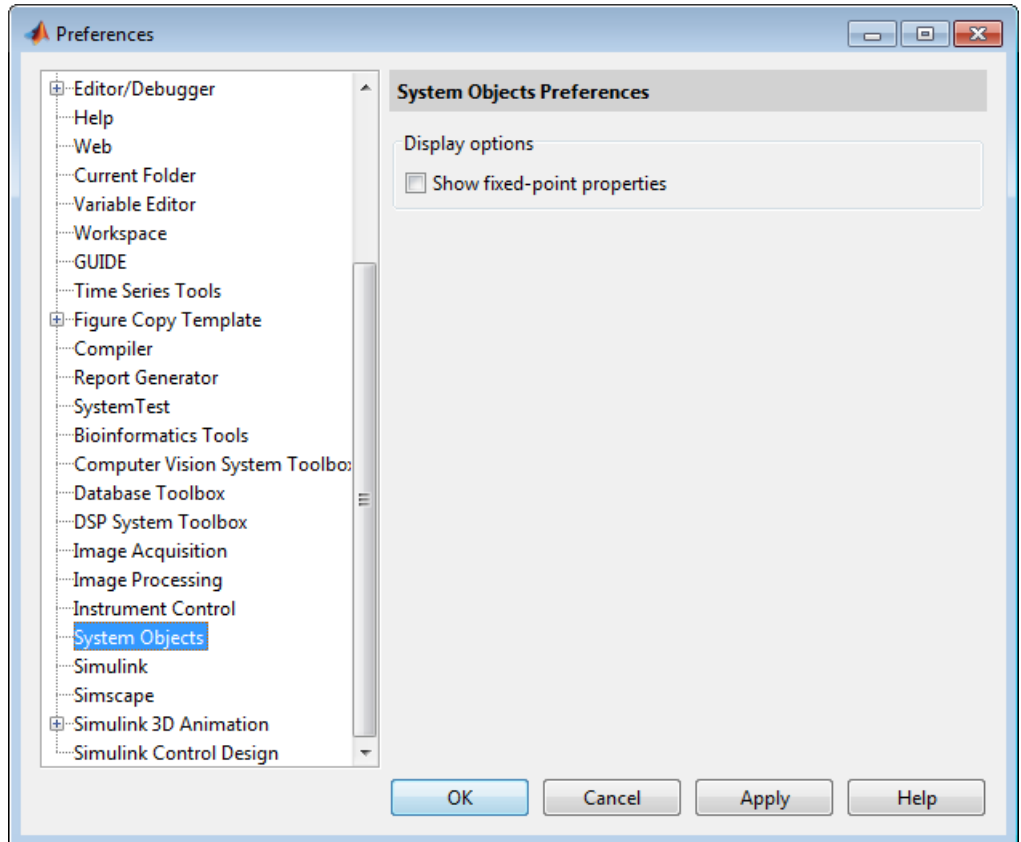
Object	Description
<code>dsp.IDCT</code>	Compute inverse discrete cosine transform (IDCT) of input
<code>dsp.IFFT</code>	Compute inverse fast Fourier transform (IFFT) of input

Display Fixed-Point Properties for System Objects

You can control whether the software displays fixed-point properties with either of the following commands:

- `matlab.system.ShowFixedPointProperties`
- `matlab.system.HideFixedPointProperties`

at the MATLAB command line. These commands set the **Show fixed-point properties** display option. You can also set the display option directly via the MATLAB preferences dialog box. Select **File > Preferences** on the MATLAB desktop, and then select **System Objects**. Finally, select or deselect **Show fixed-point properties**.



If an object supports fixed-point data processing, its fixed-point properties are active regardless of whether they are displayed or not.

Set System Object Fixed-Point Properties

A number of 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 “Change a System Object Property”. You also use the Fixed-Point Toolbox `numericity` 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)` you must set `OutputDataType` to `'Custom'`.

Note System objects do not support fixed-point word lengths greater than 128 bits.

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, you must first set `FullPrecisionOverride` to `false`.

Specify Fixed-Point Attributes for Blocks

In this section...

“Fixed-Point Block Parameters” on page 8-28

“Specify System-Level Settings” on page 8-31

“Inherit via Internal Rule” on page 8-32

“Select and Specify Data Types for Fixed-Point Blocks” on page 8-43

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.

Main Data Types

Fixed-point operational parameters

Rounding mode: Overflow mode:

Floating-point inheritance takes precedence over the settings in the 'Data Type' column below. When the block input is floating point, all block data types match the input.

	Data Type	Assistant	Minimum	Maximum
Product output:	<input type="text" value="Inherit: Inherit via internal rule"/>	<input type="text" value=">>"/>		
Accumulator:	<input type="text" value="Inherit: Inherit via internal rule"/>	<input type="text" value=">>"/>		
Output:	<input type="text" value="Inherit: Same as first input"/>	<input type="text" value=">>"/>	<input type="text" value="[]"/>	<input type="text" value="[]"/>

Lock data type settings against changes by the fixed-point tools

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 8-8 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 8-7 for more information on the available options.

Fixed-Point Data Type Parameter	Description
Intermediate Product	<p>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.</p> <p>See the reference page of a specific block to learn about the intermediate product data type for that block.</p>
Product Output	<p>Specifies the data type and scaling of the product output for fixed-point blocks that must compute multiplication results.</p> <p>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 8-13 for more information on complex fixed-point multiplication in System toolbox software.</p>
Accumulator	<p>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).</p> <p>See the reference page of a specific block for details on the accumulator data type of that block.</p>
Output	<p>Specifies the output data type and scaling for blocks.</p>

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:

“Using the 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 Simulink Fixed Point 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 8-32
- “Internal Rule for Product Data Types” on page 8-33
- “Internal Rule for Output Data Types” on page 8-33
- “The Effect of the Hardware Implementation Pane on the Internal Rule” on page 8-34
- “Internal Rule Examples” on page 8-35

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 8-34 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 8-34 for more information.

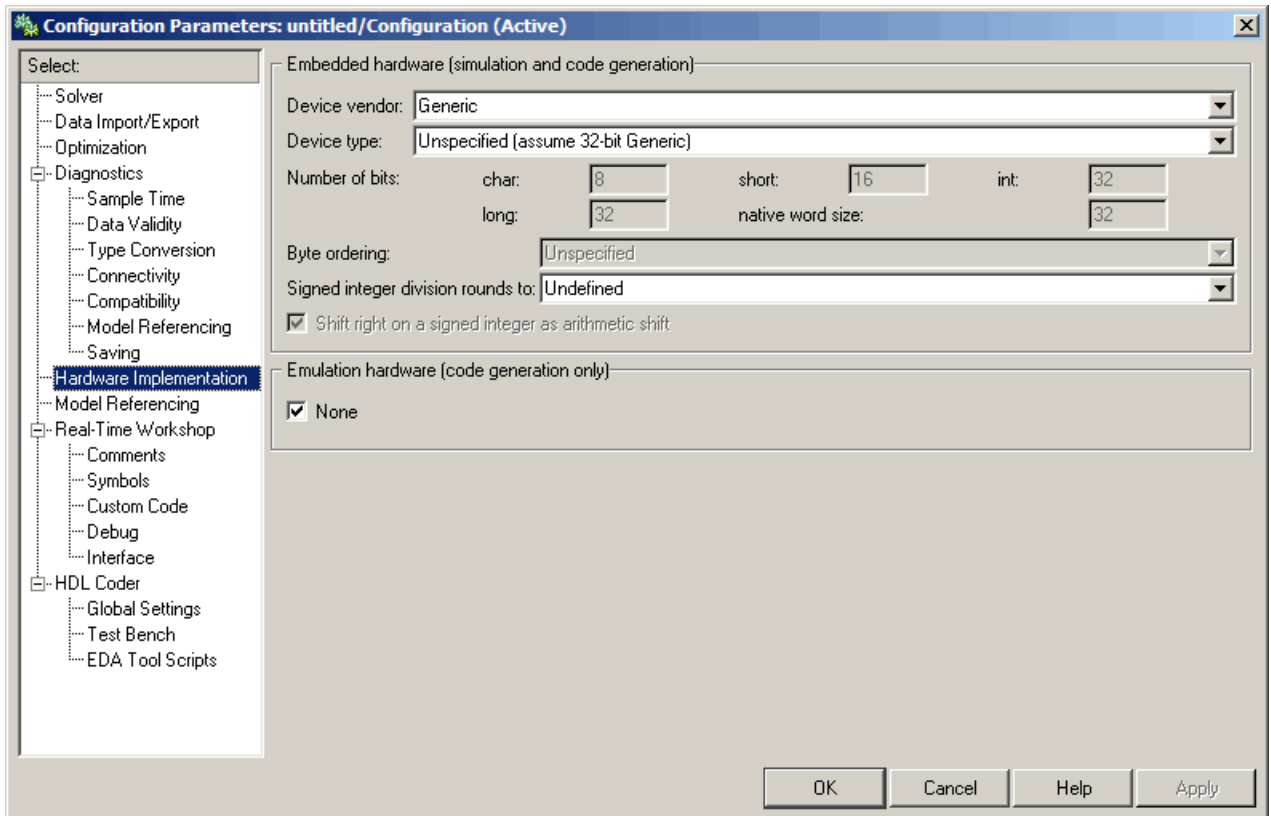
Internal Rule for Output Data Types

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 8-34.

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.

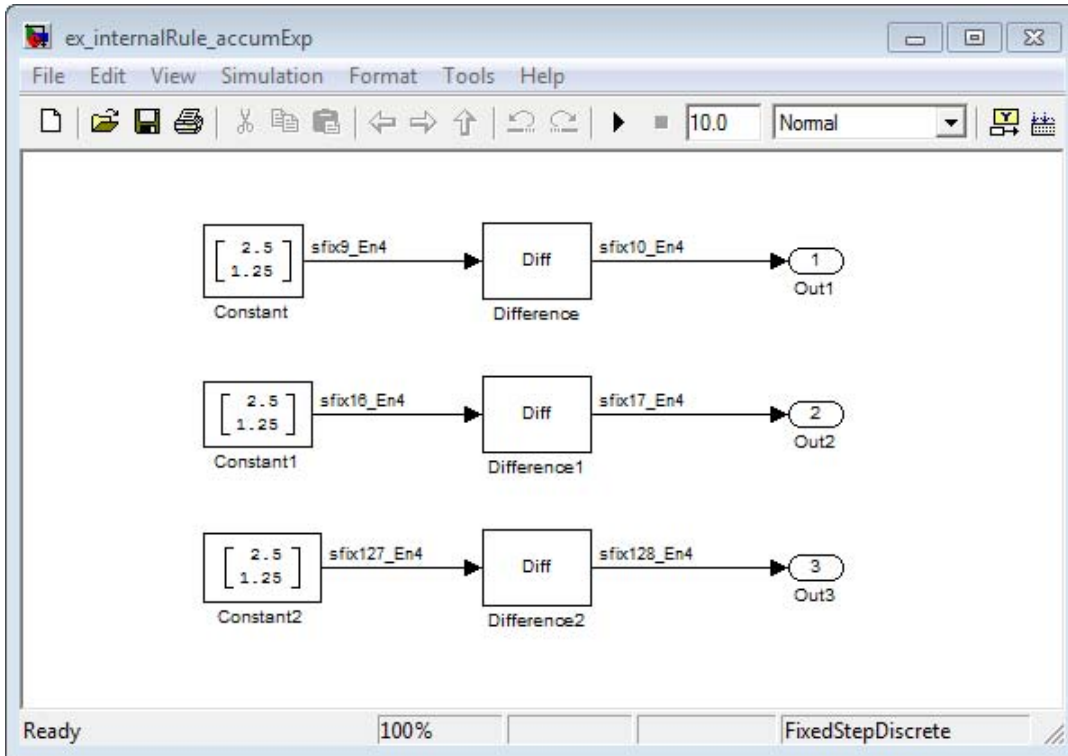
Other targets. For all targets other than ASIC/FPGA, the ideal, full-precision word length calculated by the internal rule is rounded up to the next available word length of the target. The calculated ideal fraction length is used, keeping the least-significant bits.

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:

$$WL_{ideal\ accumulator} = WL_{input\ to\ accumulator} + \text{floor}(\log_2(\text{number of accumulations})) + 1$$

$$WL_{ideal\ accumulator} = 9 + \text{floor}(\log_2(1)) + 1$$

$$WL_{ideal\ accumulator} = 9 + 0 + 1 = 10$$

$$WL_{ideal\ accumulator1} = WL_{input\ to\ accumulator1} + \text{floor}(\log_2(\text{number of accumulations})) + 1$$

$$WL_{ideal\ accumulator1} = 16 + \text{floor}(\log_2(1)) + 1$$

$$WL_{ideal\ accumulator1} = 16 + 0 + 1 = 17$$

$$WL_{ideal\ accumulator2} = WL_{input\ to\ accumulator2} + \text{floor}(\log_2(\text{number of accumulations})) + 1$$

$$WL_{ideal\ accumulator2} = 127 + \text{floor}(\log_2(1)) + 1$$

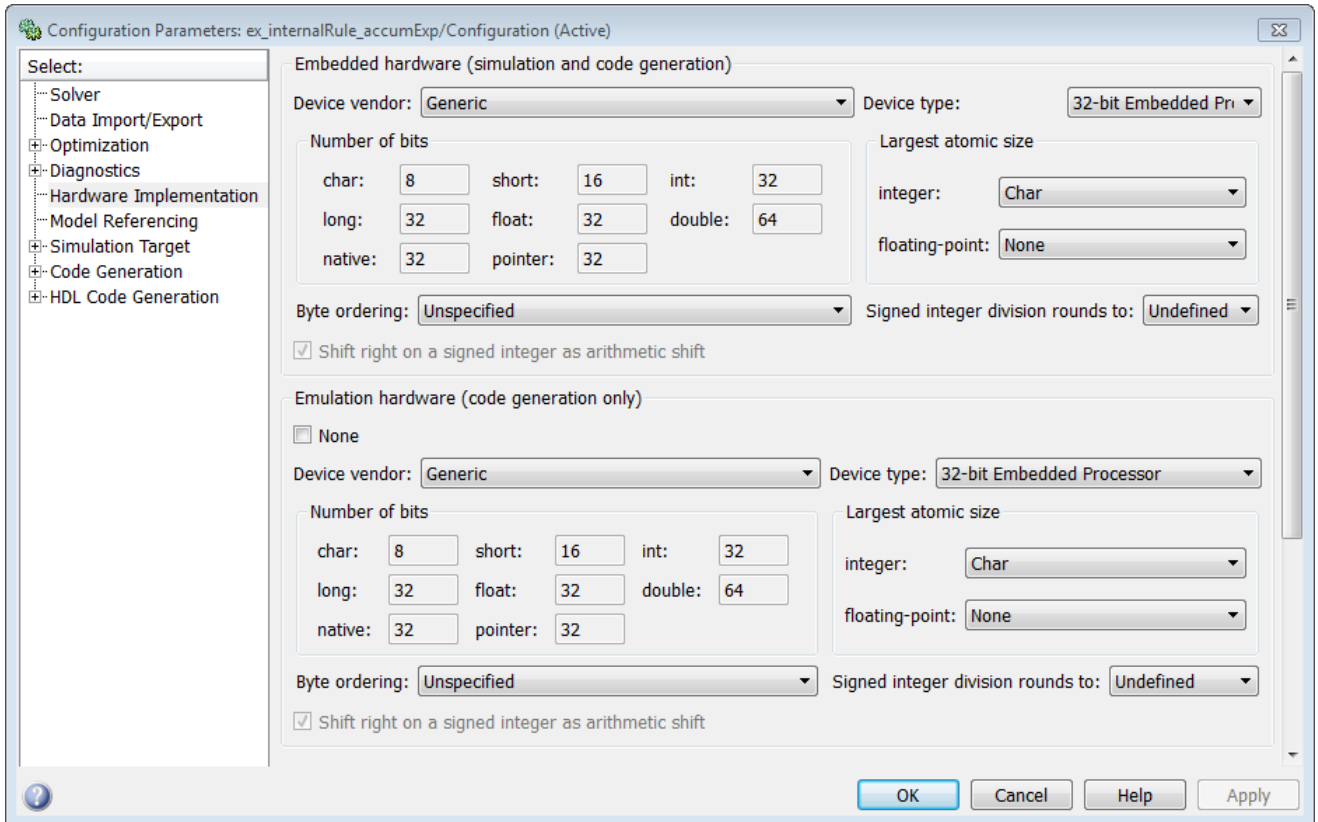
$$WL_{ideal\ accumulator2} = 127 + 0 + 1 = 128$$

Calculate the full-precision fraction length, which is the same for each Matrix Sum block in this example:

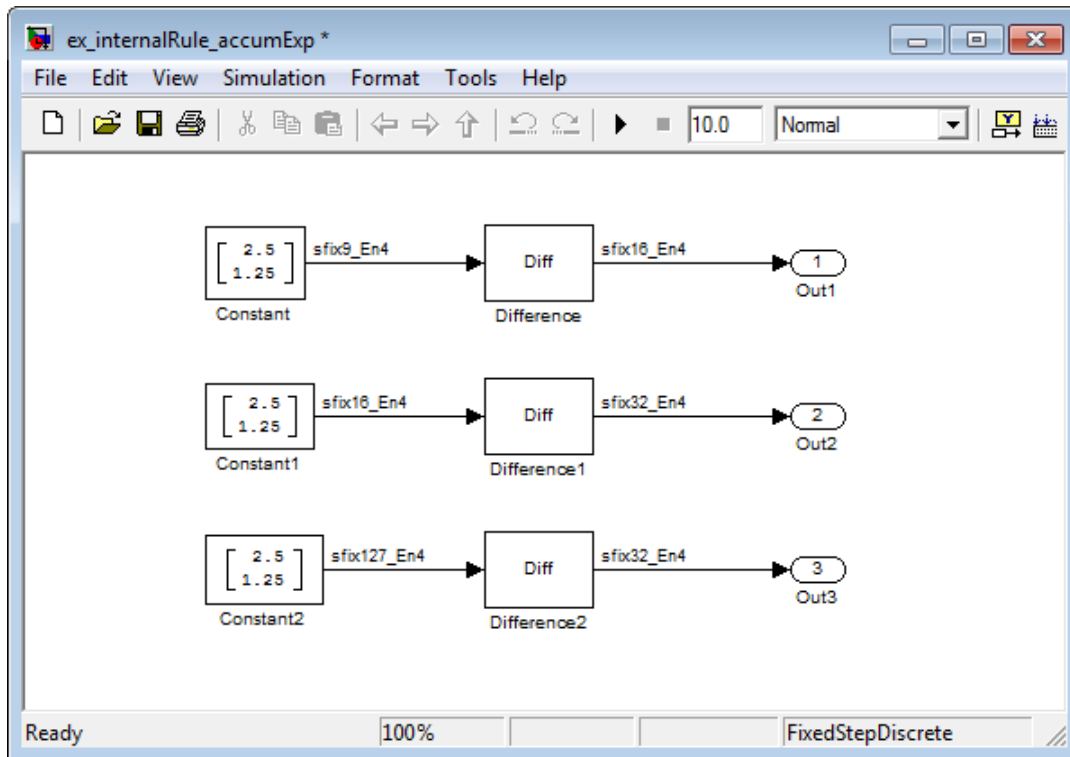
$$FL_{ideal\ accumulator} = FL_{input\ to\ accumulator}$$

$$FL_{ideal\ accumulator} = 4$$

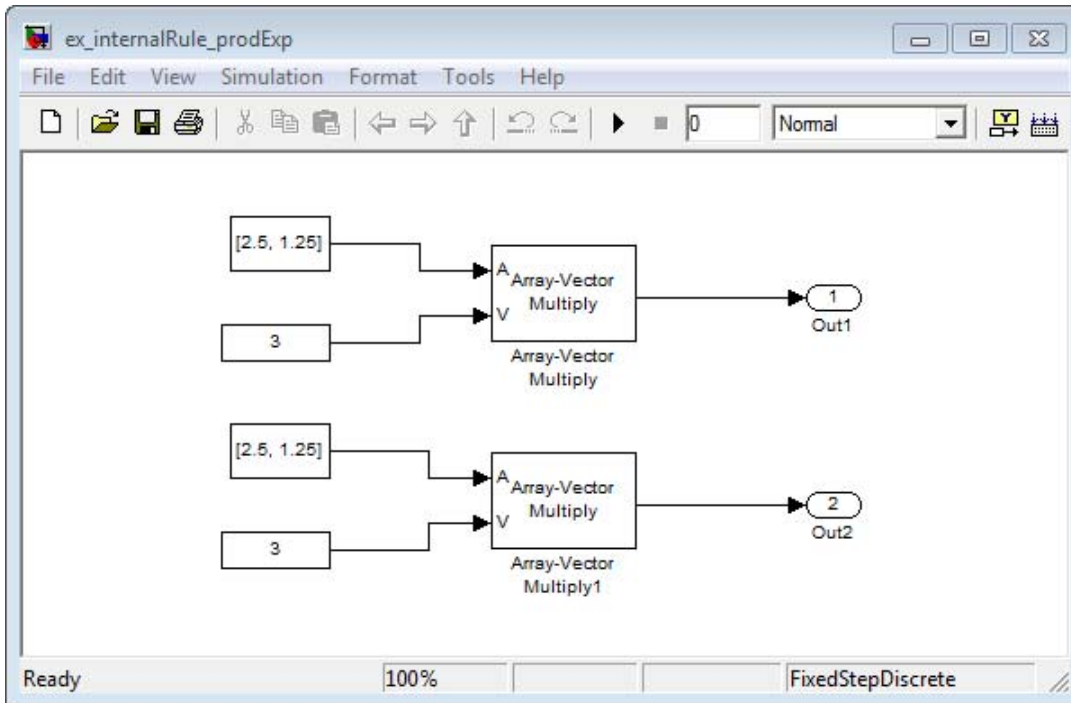
Now change the **Device type** parameter in the **Hardware Implementation** pane of the Configuration Parameters dialog box to 32 bit Embedded Processor, by changing the parameters as shown in the following figure.



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:

$$WL_{ideal\ product} = WL_{input\ a} + WL_{input\ b}$$

$$WL_{ideal\ product} = 7 + 5 = 12$$

$$WL_{ideal\ product1} = WL_{input\ a} + WL_{input\ b}$$

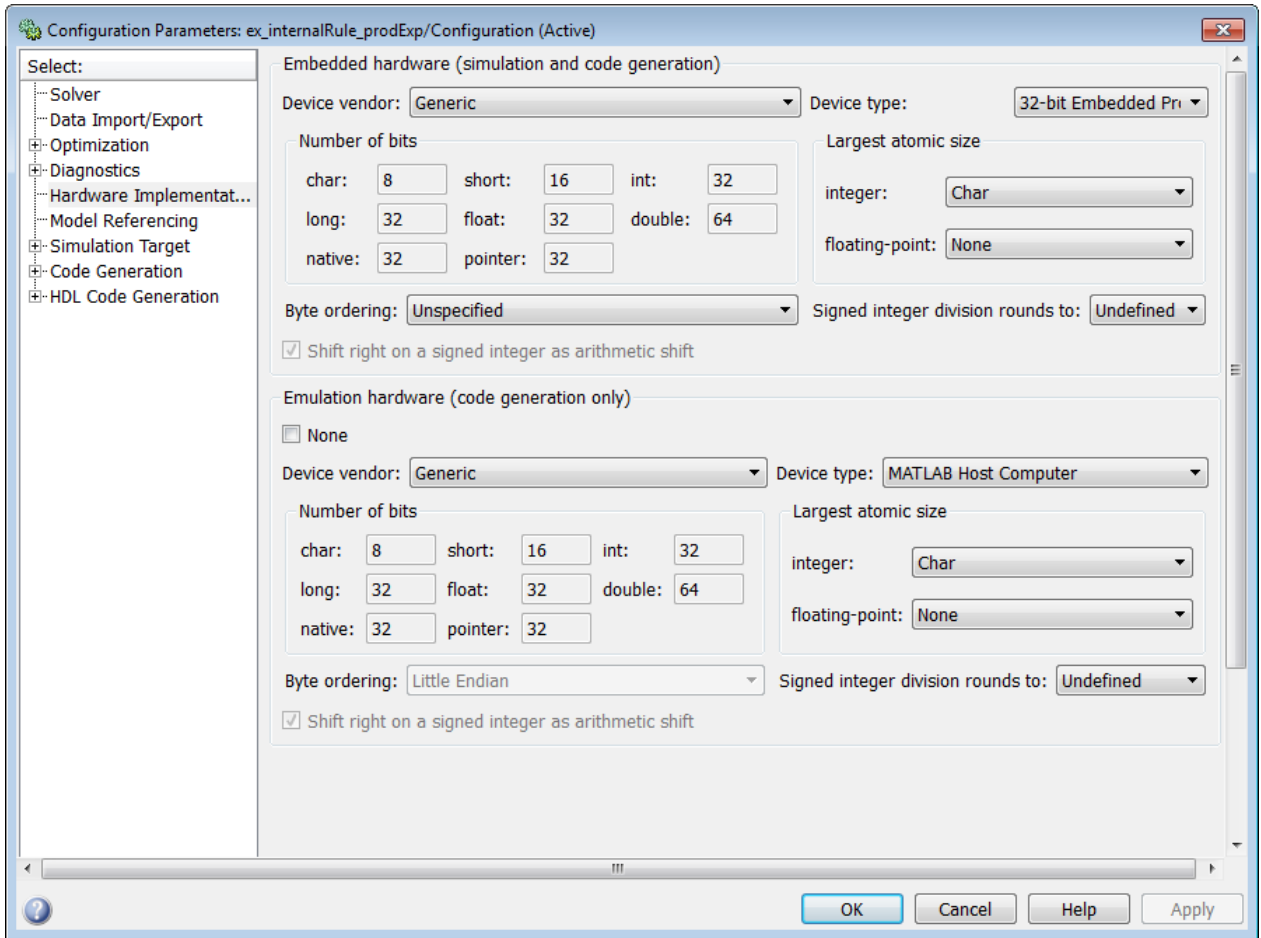
$$WL_{ideal\ product1} = 16 + 15 = 31$$

Calculate the full-precision fraction length, which is the same for each Array-Vector Multiply block in this example:

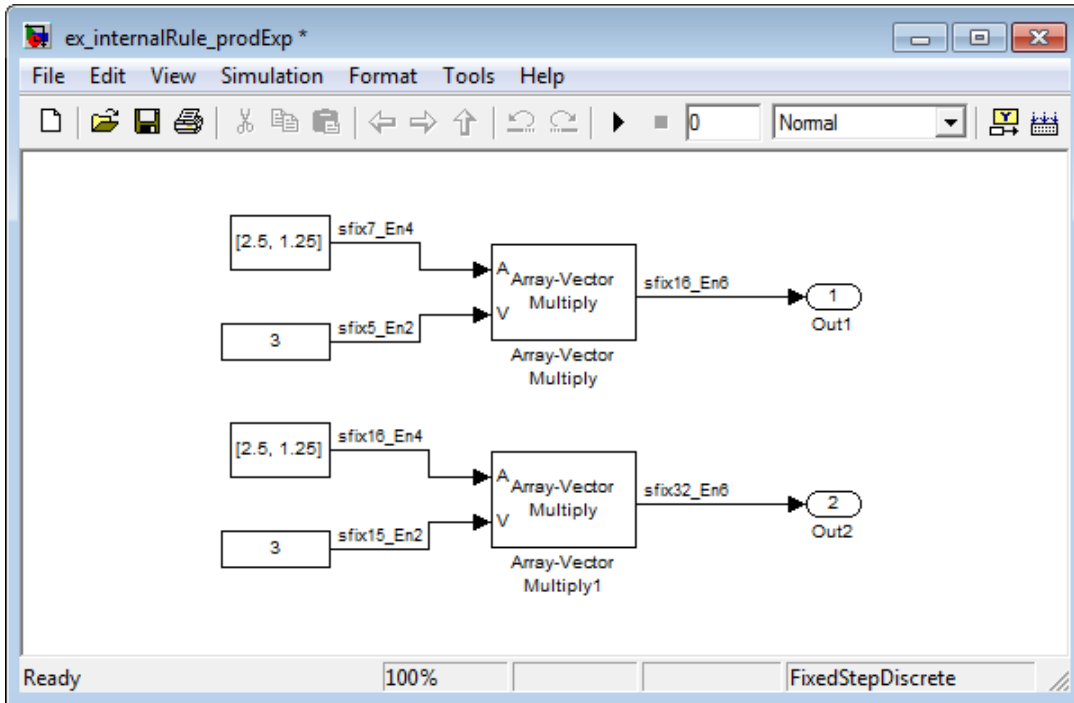
$$FL_{ideal\ accumulator} = FL_{input\ to\ accumulator}$$

$$FL_{ideal\ accumulator} = 4$$

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.



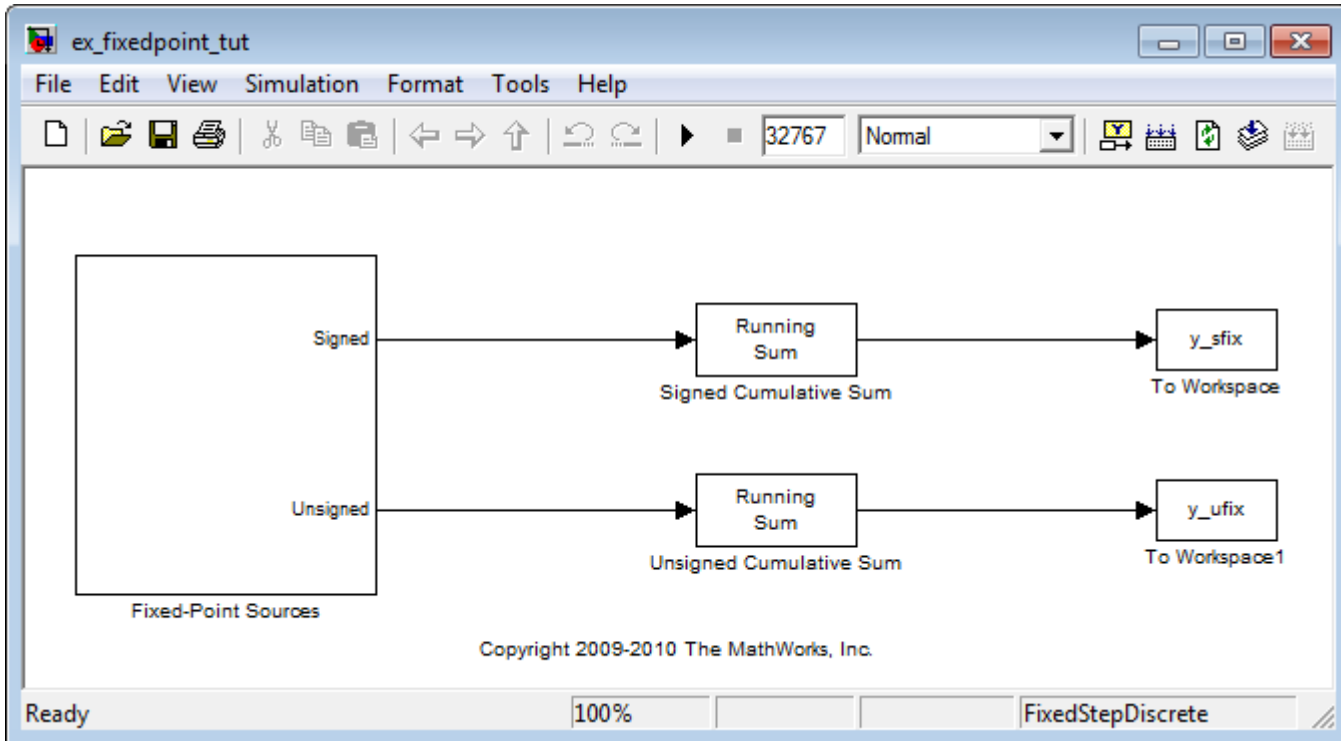
Select and Specify Data Types for Fixed-Point Blocks

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 8-43
- “Use Data Type Override to Find a Floating-Point Benchmark” on page 8-48
- “Use the Fixed-Point Tool to Propose Fraction Lengths” on page 8-49
- “Examine the Results and Accept the Proposed Scaling” on page 8-49

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:

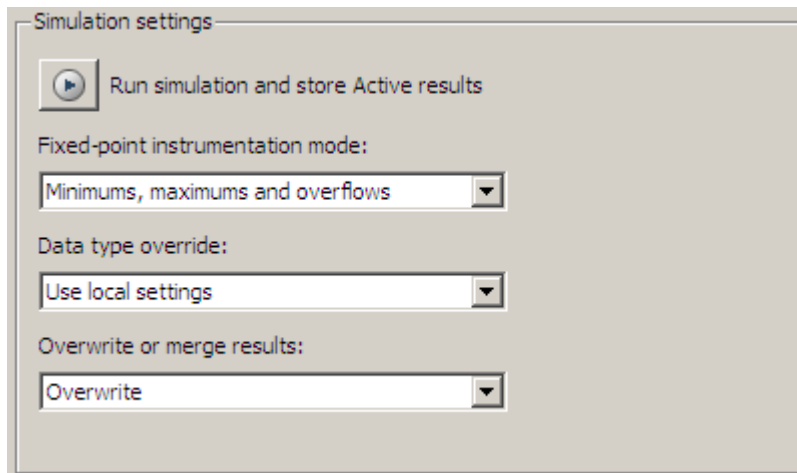
```
Warning: Overflow occurred. This originated from
'ex_fixedpoint_tut/Signed Cumulative Sum'.
```

```
Warning: Overflow occurred. This originated from
```


'ex_fixedpoint_tut/Unsigned Cumulative Sum'.

According to these warnings, overflow occurs in both Cumulative Sum blocks. You can control the display of these warnings using the “Configuration Parameters Dialog Box”.

- 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 **Run simulation and store active results** button in the **Simulation settings** pane.



- 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.

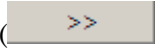
- **SimMin** — The smallest representable value achieved during simulation for each logged signal.
- **SimMax** — The largest representable value achieved during simulation for each logged signal.
- **OverflowWraps** — The number of overflows that wrap during simulation.

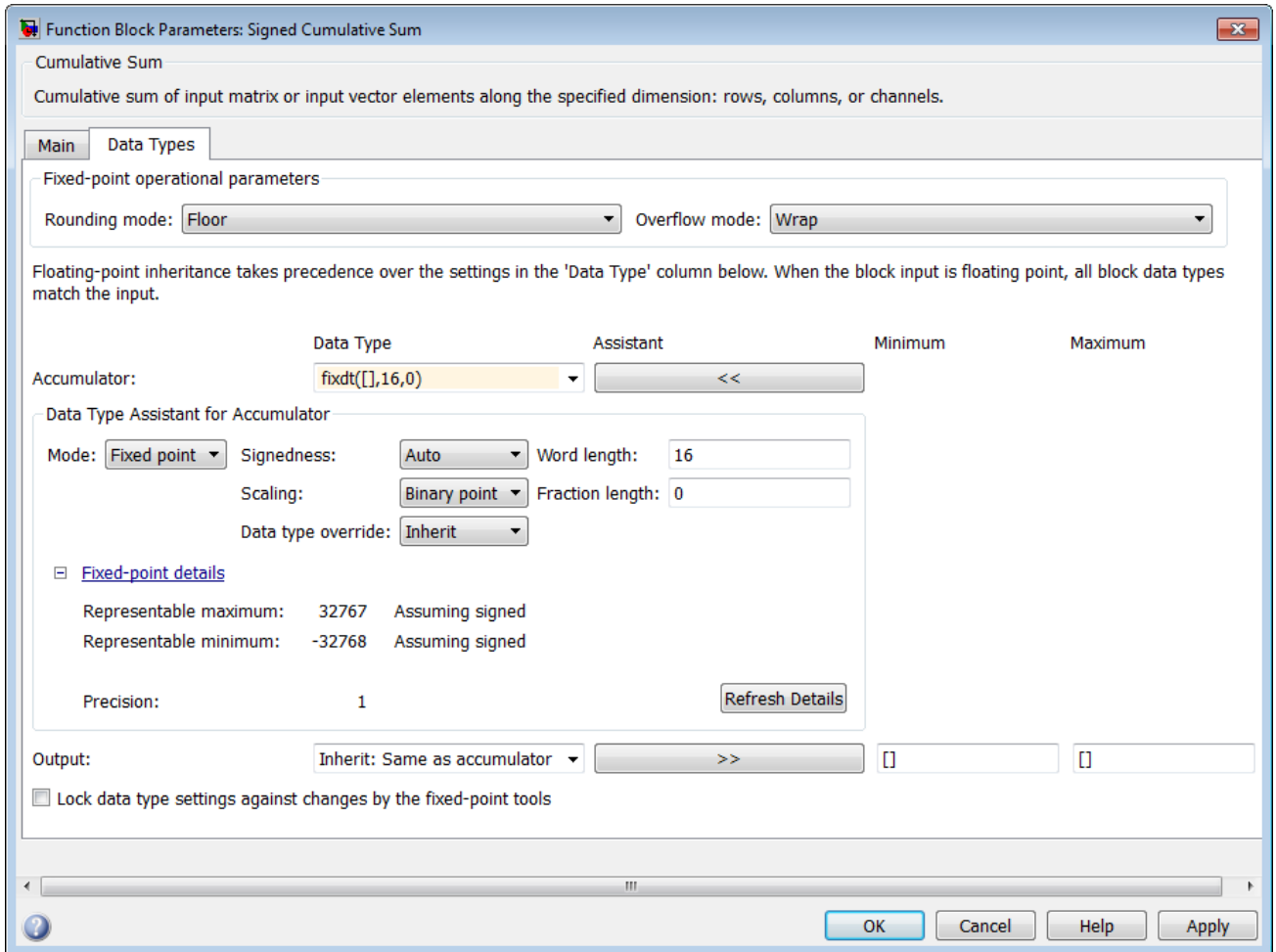
For more information on each of the columns in this table, see the “Contents Pane” section of the Simulink `fxptdlg` function reference page.

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:
 - a Right-click the Signed Cumulative Sum: Accumulator row in the Fixed-Point Tool pane, and select **Highlight Block In Model**.
 - b Double-click the block in the model, and select the **Data Types** pane of the dialog box.
 - c Open the **Data Type Assistant for Accumulator** by clicking the Assistant button () in the Accumulator data type row.
 - d 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.



- e 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.
- f Click **OK** on the block dialog box to save your changes and close the window.
- g 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 (). 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 ().
- 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” on page 8-51
“Vector Quantizers” on page 8-58

Scalar Quantizers

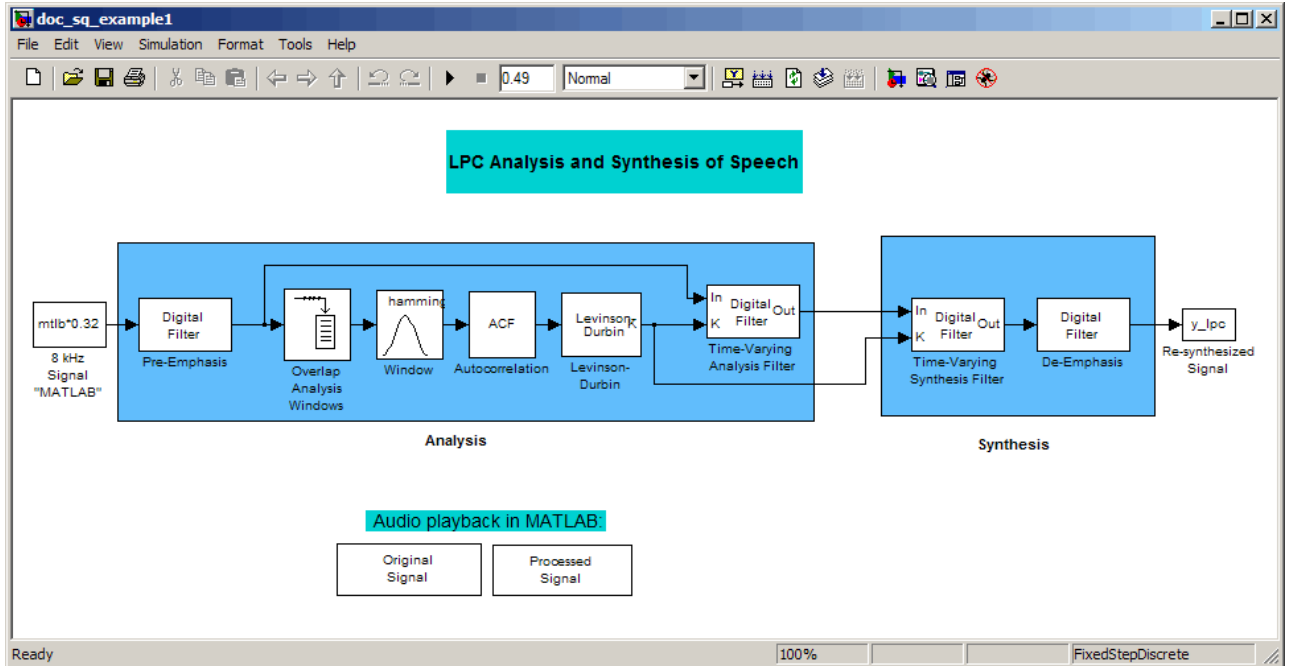
- “Analysis and Synthesis of Speech” on page 8-51
- “Identify Your Residual Signal and Reflection Coefficients” on page 8-53
- “Create a Scalar Quantizer” on page 8-54

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 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 preemphasizes 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 preemphasized 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 deemphasis filter. Note that the deemphasis filter is the inverse of the preemphasis 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, you need to 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.

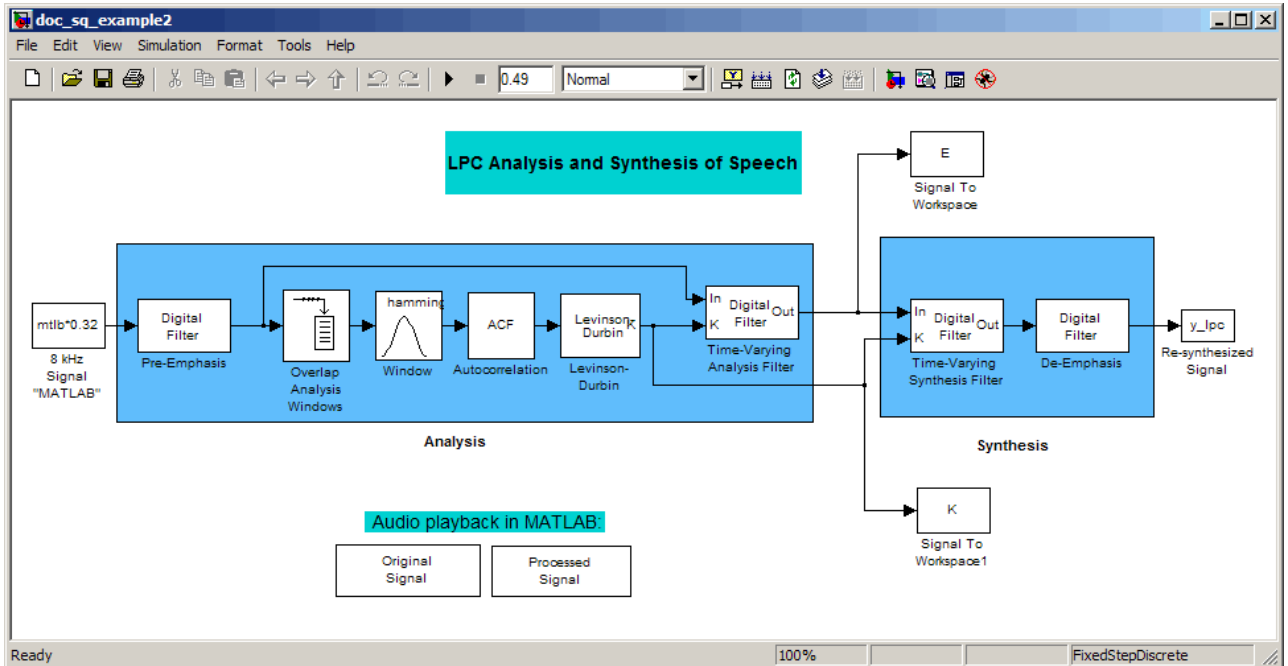
Identify Your Residual Signal and Reflection Coefficients

In the previous topic, “Analysis and Synthesis of Speech” on page 8-51, 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_example1` at the MATLAB command line.
- 2** Save the model file as `ex_sq_example2` in your working folder.
- 3** From the Sinks library, click-and-drag two Signal To Workspace blocks into your model.
- 4** Connect the output of the Levinson-Durbin block to one of the Signal To Workspace blocks.
- 5** Double-click this Signal 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 Signal To Workspace block.

- 7 Double-click this Signal To Workspace block and set the **Variable name** parameter to **E**. Click **OK**.

You 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 8-53 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.

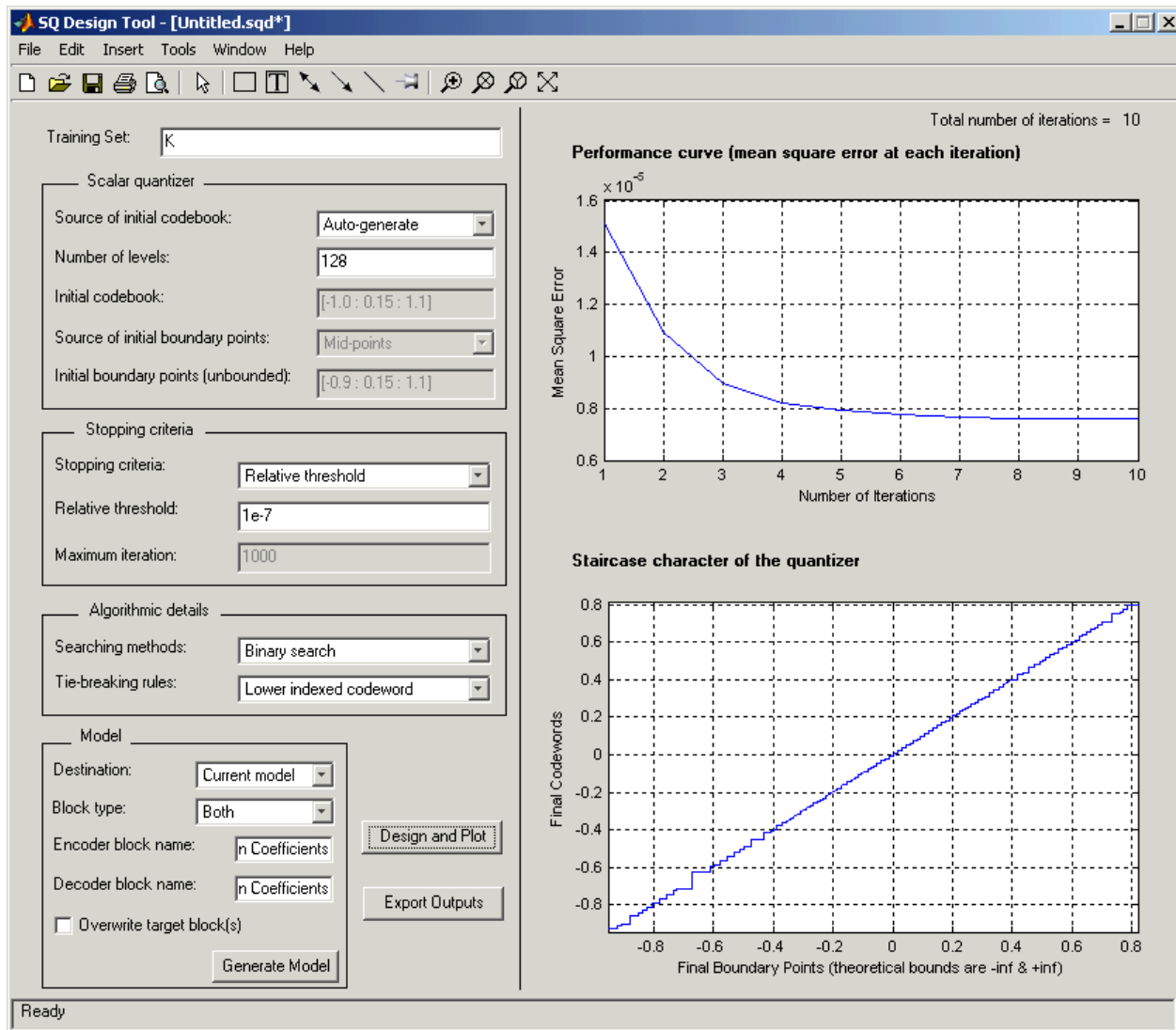
7 For the **Encoder block name** parameter, enter SQ Encoder - Reflection Coefficients.

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 .

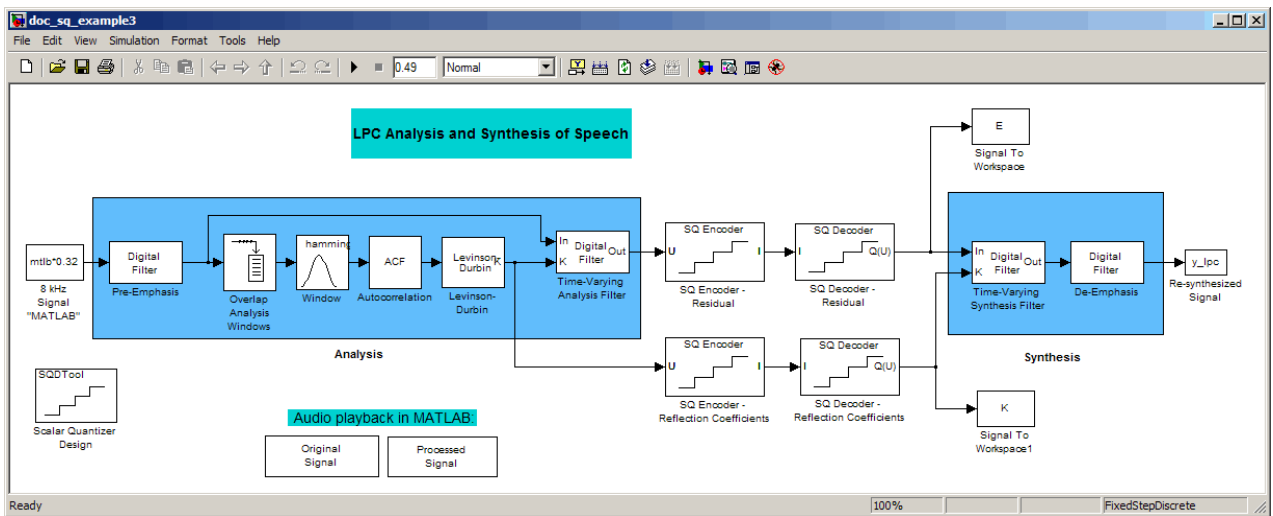
- 13 Repeat steps 5 to 11 for the variable E , which represents the residual signal you want to quantize. In steps 6 and 7, name your blocks SQ Encoder - Residual and SQ Decoder - Residual.

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 8-58.

Vector Quantizers

- “Build Your Vector Quantizer Model” on page 8-58
- “Configure and Run Your Model” on page 8-60

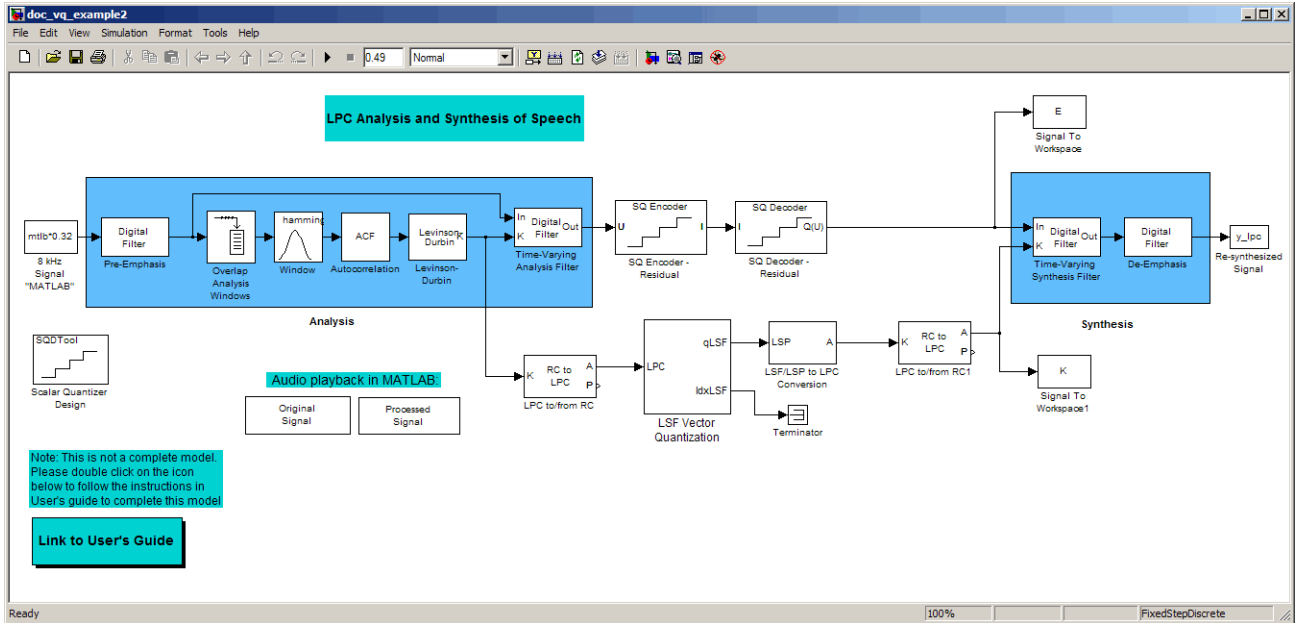
Build Your Vector Quantizer Model

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 8-54 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.

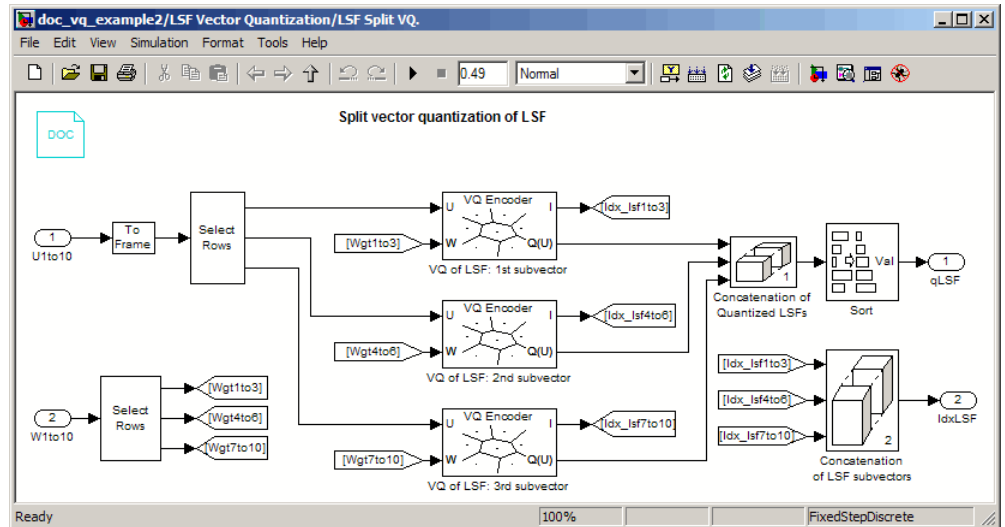
Configure and Run Your Model

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 8-58 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.

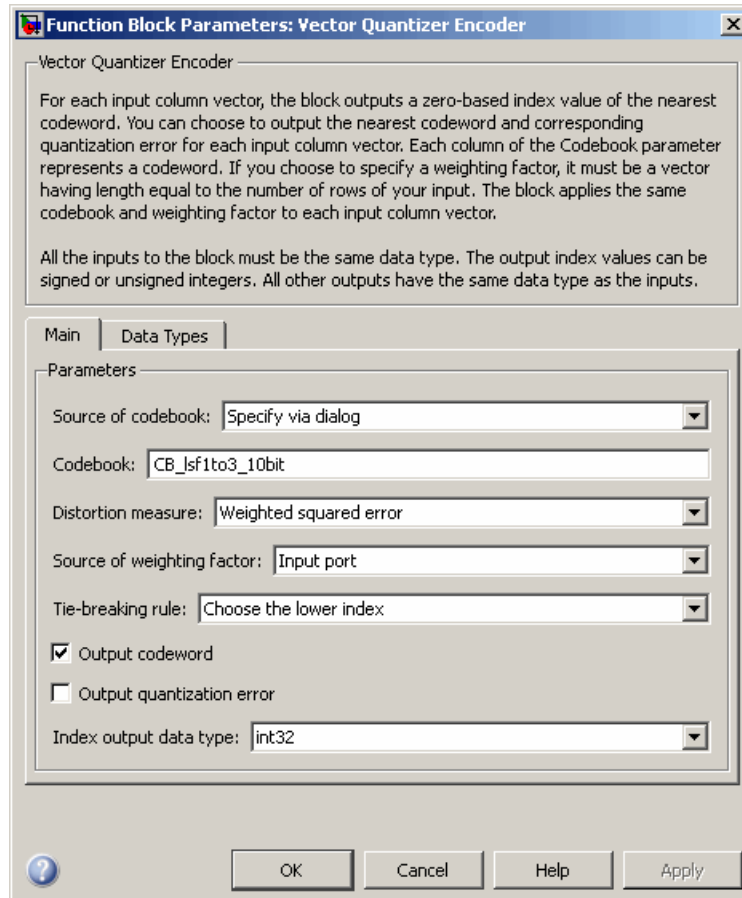
The subsystem opens, and you see the three Vector Quantizer Encoder blocks used to implement the split vector quantization method.



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

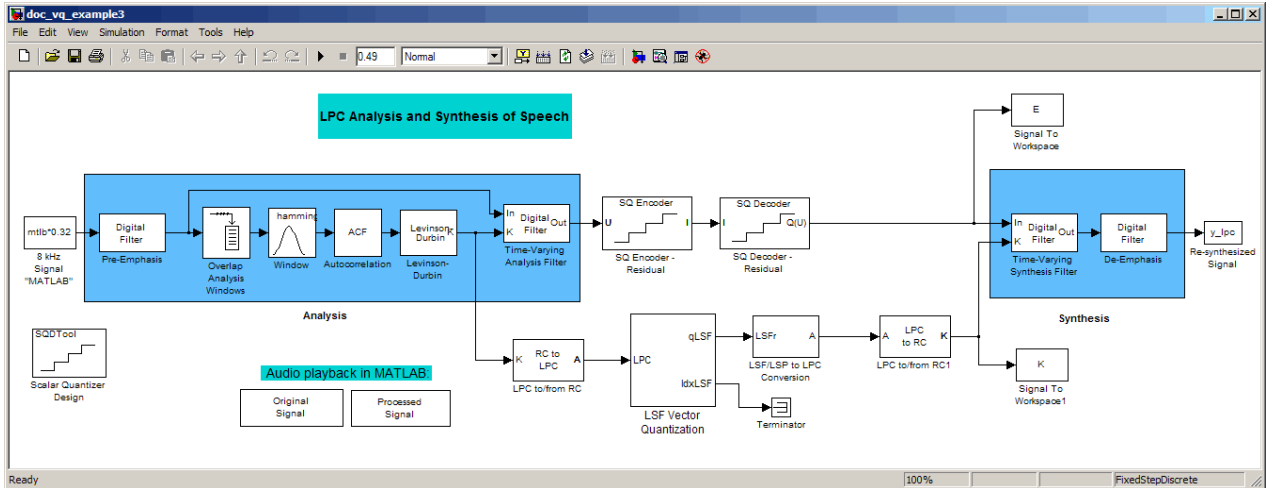
and 7 to 10 of the LSF vector, respectively. Therefore, it takes 30 bits to quantize all three subvectors.

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 $[(80 \text{ residual samples/frame}) \cdot (7 \text{ bits/sample}) + (30 \text{ bits/frame})] \cdot (100 \text{ frames/second})$, 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 8-65

“Data Types for Filter Functions” on page 8-65

“Convert a Filter from Floating Point to Fixed Point in MATLAB” on page 8-67

“Create an FIR Filter Using Integer Coefficients” on page 8-75

“Fixed-Point Filtering in Simulink” on page 8-92

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 math, see “Fixed-Point Concepts” in “Fixed-Point Toolbox” documentation.

Data Types for Filter Functions

- “Data Type Support” on page 8-65
- “Fixed Data Type Support” on page 8-66
- “Single Data Type Support” on page 8-66

Data Type Support

There are three different data types supported in DSP System Toolbox software:

- Fixed — Requires Fixed Point Toolbox and is supported by packages listed in “Fixed Data Type Support” on page 8-66.
- 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 8-66.

Fixed Data Type Support

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

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

To set the filter’s arithmetic property:

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

Single Data Type Support

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

Convert a Filter from Floating Point to Fixed Point in MATLAB

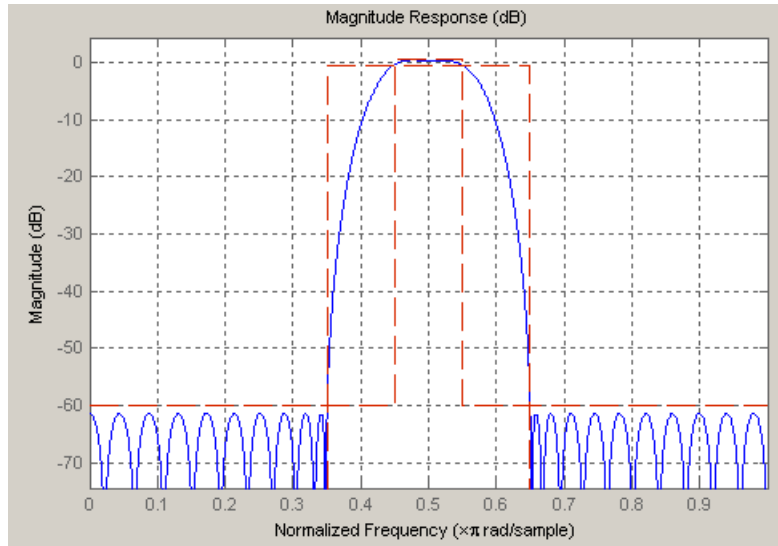
- “Process Overview” on page 8-67
- “Design the Filter” on page 8-67
- “Quantize the Coefficients” on page 8-68
- “Dynamic Range Analysis” on page 8-71
- “Compare Magnitude Response and Magnitude Response Estimate” on page 8-72

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.

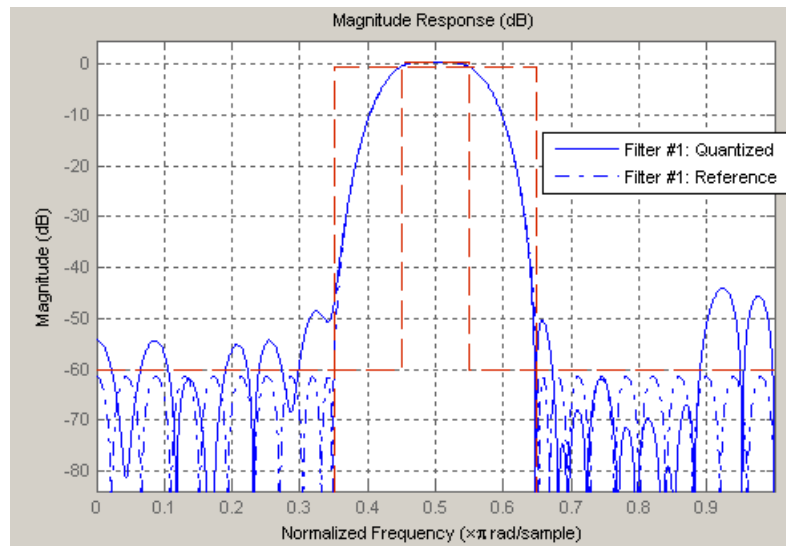
Quantize the Coefficients

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.

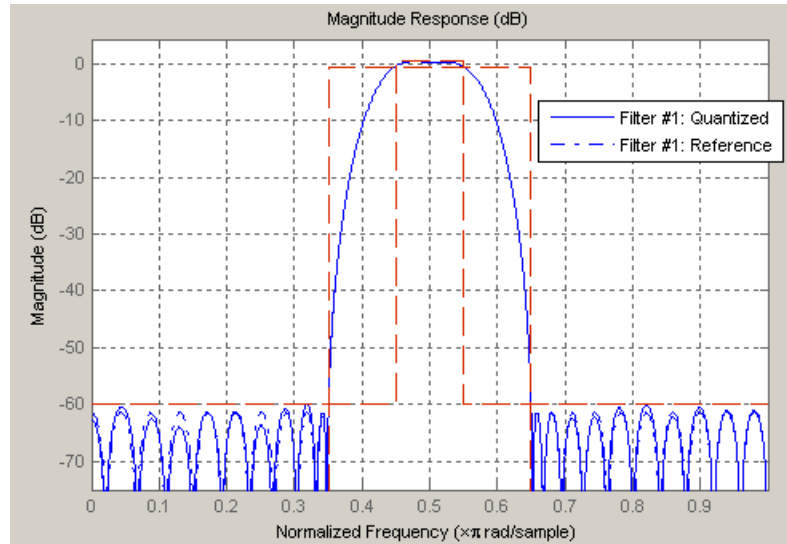


As the figure shows, the filter design constraints are not met. The attenuation is not complete, and there is noise at the edges of the stopbands. You can experiment with different coefficient word lengths if you like. For this example, however, the word length of 12 is sufficient.

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');
```

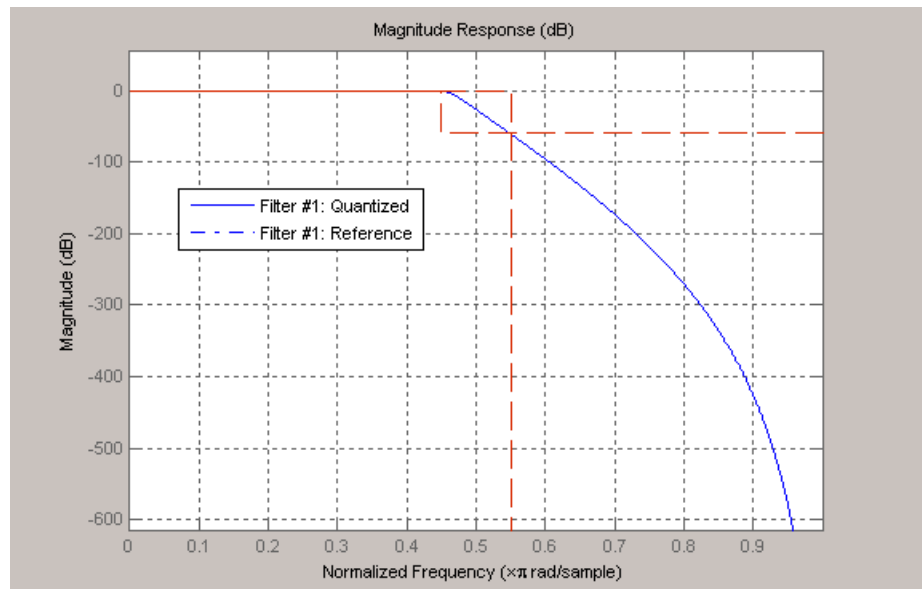
Now set the arithmetic to fixed-point.

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

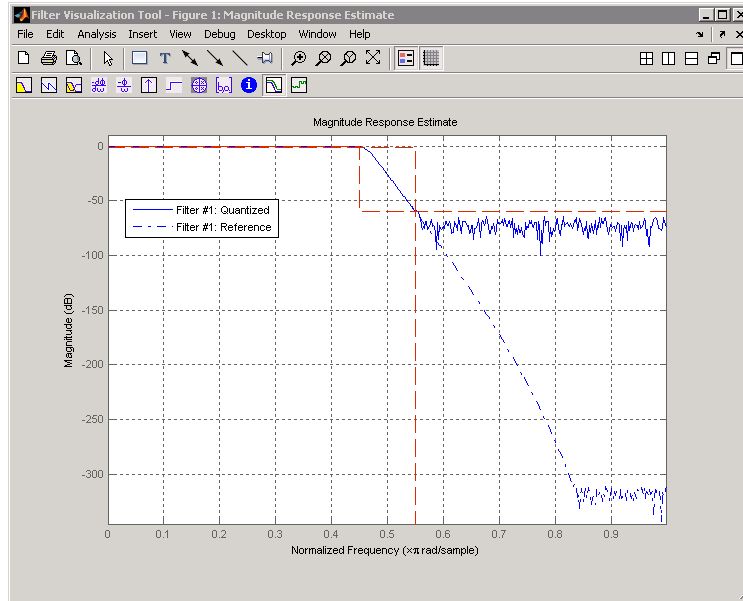
Open the filter using fvtool.

```
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 F_s . For more information about analyzing filters in this way, refer to the section titled Analyzing Filters with a Noise-Based Method in the User Guide.

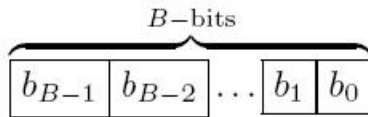
For more information, refer to McClellan, et al., Computer-Based Exercises for Signal Processing Using MATLAB 5, Prentice-Hall, 1998. See Project 5: Quantization Noise in Digital Filters, page 231.

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:

$$\boxed{b_{B-1}} \boxed{b_{B-2}} \dots \boxed{b_1} \boxed{b_0}$$

$$-2^{B-1-L} \ 2^{B-2-L} \qquad \qquad \qquad 2^{1-L} \ 2^{-L}$$

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:

$$\begin{array}{cccc} \boxed{b_3} & \boxed{b_2} & \boxed{b_1} & \boxed{b_0} \\ -2^{3-L} & 2^{2-L} & 2^{1-L} & 2^{-L} \end{array}$$

$$\begin{array}{cc} \boxed{b_3} & \boxed{b_2} \\ -2^{3-L} & 2^{2-L} \end{array}$$

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 $\{-32768, -32767, \dots, -1, 0, 1, \dots, 32766, 32767\}$.

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 $\{-32768, -32512, \dots, -256, 0, 256, \dots, 32256, 32512\}$.

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

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

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

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

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

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

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

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

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:

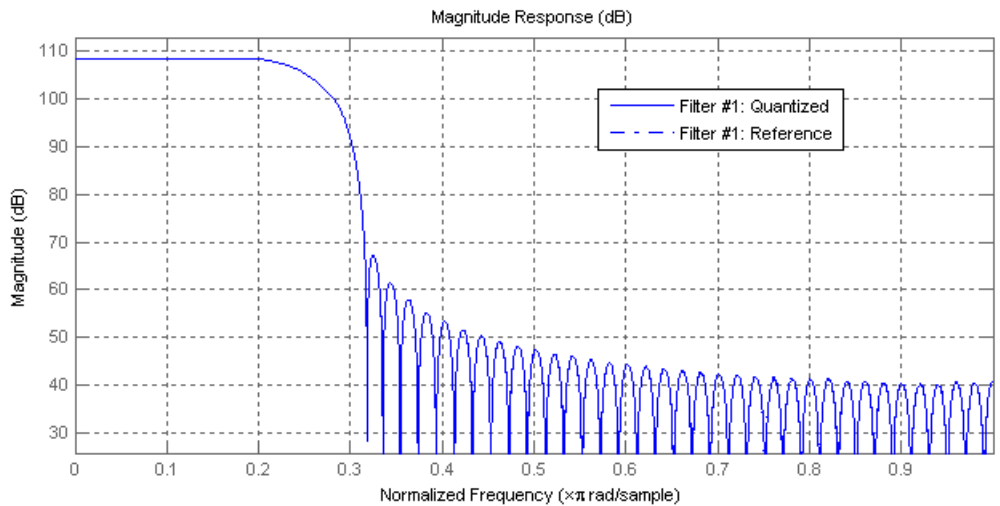
```
all(h.Numerator == round(h.Numerator))
```

```
ans =
```

```
1
```

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

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



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

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

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:

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

Here `y_sc` is a full precision output, meaning that no bits have been discarded in the computation. This makes `y_sc` 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:

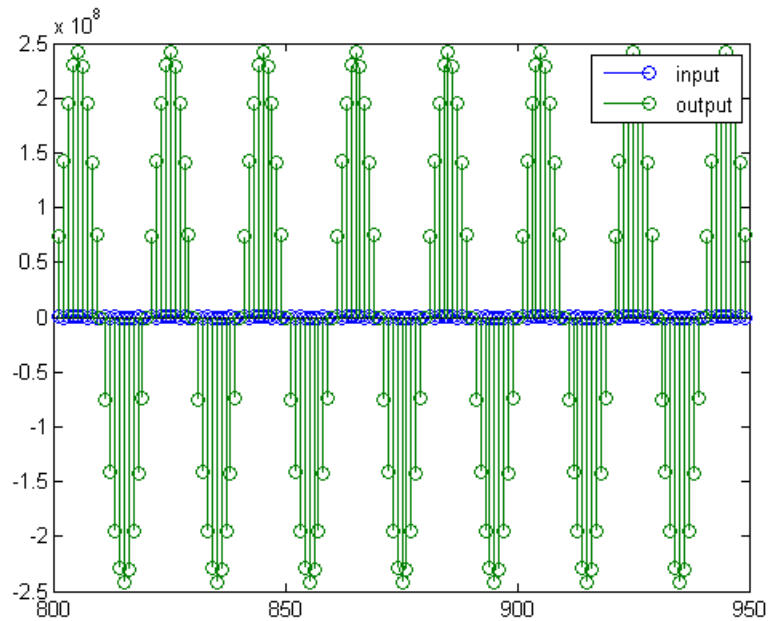
```
hd = double(h);
xd = double(xsc);
yd = filter(hd, xd);
norm(yd-double(y_sc))
```

```
ans =
```

```
0
```

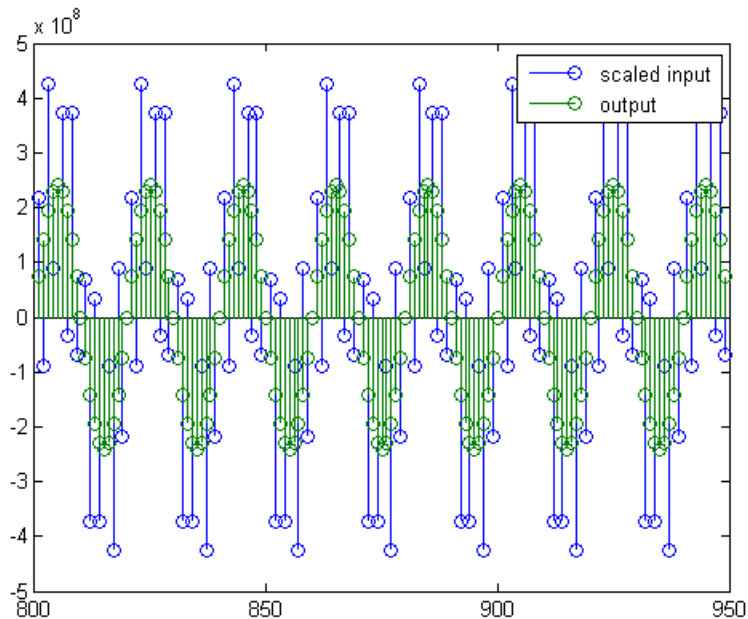
Now you can examine the output compared to the input. This example is plotting only the last few samples to minimize the effect of transients:

```
idx = 800:950;
xs_cext = double(xsc(idx));
gd = grpdelay(h, [f1 f2]);
yidx = idx + gd(1);
ys_cext = double(y_sc(yidx));
stem(n(idx)', [xs_cext, ys_cext]);
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 8-75 the 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:

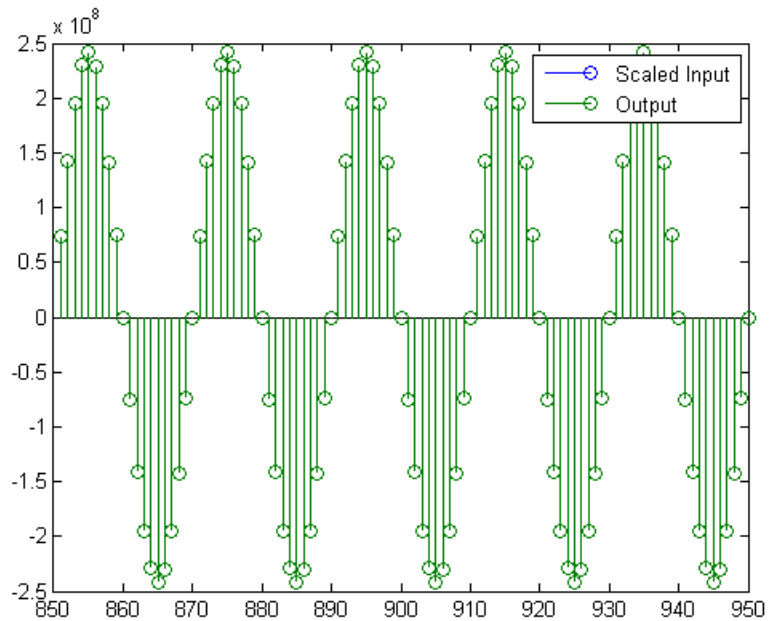
```
norm(double(yout) - double(ysc))
```

```
ans =
```

```
1.446323386867543e+005
```

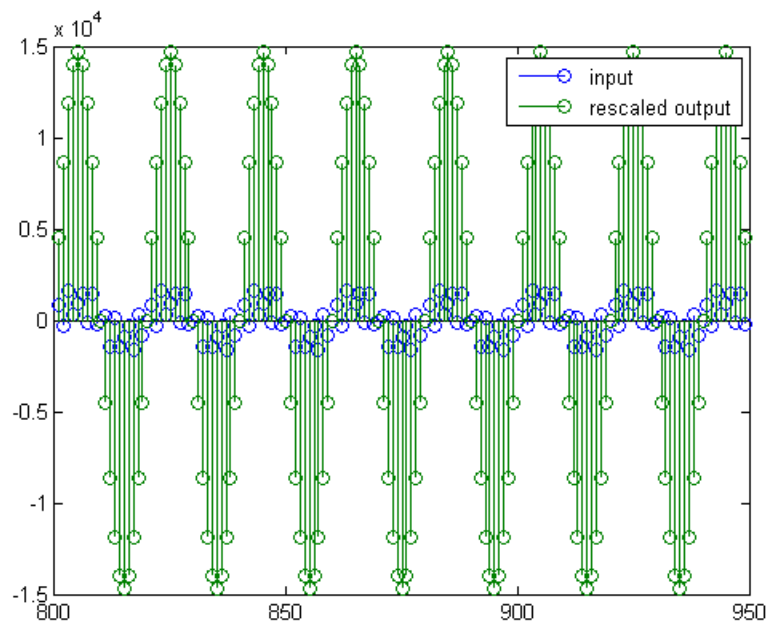
In this case the differences are hard to spot when plotting the data, as seen below:

```
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:

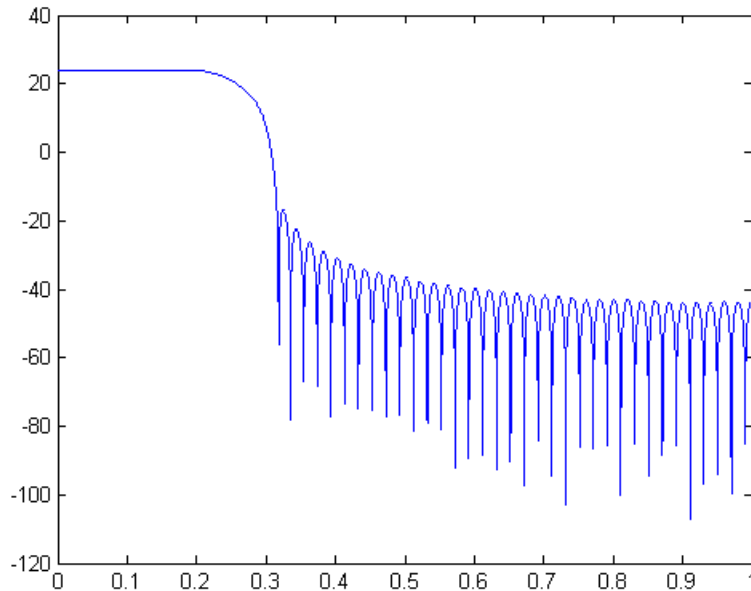
```
yri = fi(yout.int, true, 16, 0);
stem(n(idx)', [xscect, double(yri(yidx)')]);
axis([800 950 -1.5e4 1.5e4]);
legend('input', 'rescaled output');
```



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:

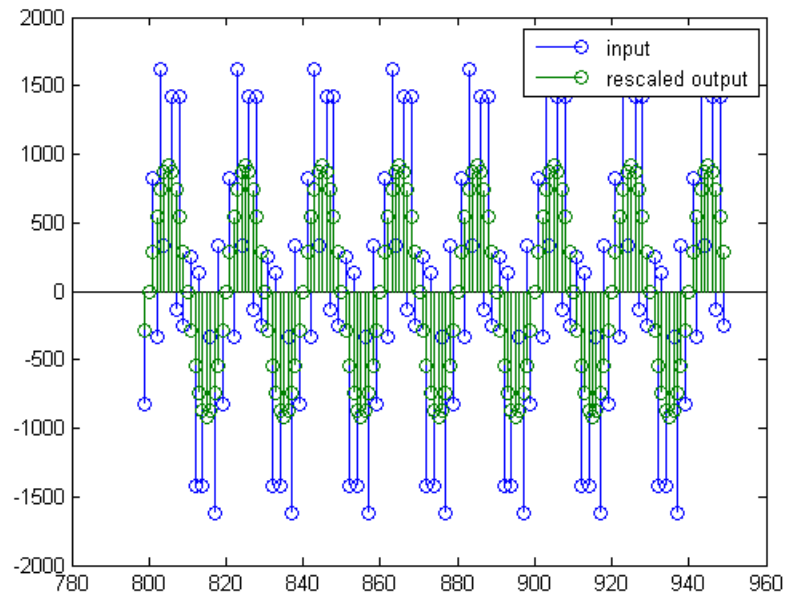
```
[H,w] = freqz(h);
plot(w/pi, 20*log10(2^(-14)*abs(H)));
```



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

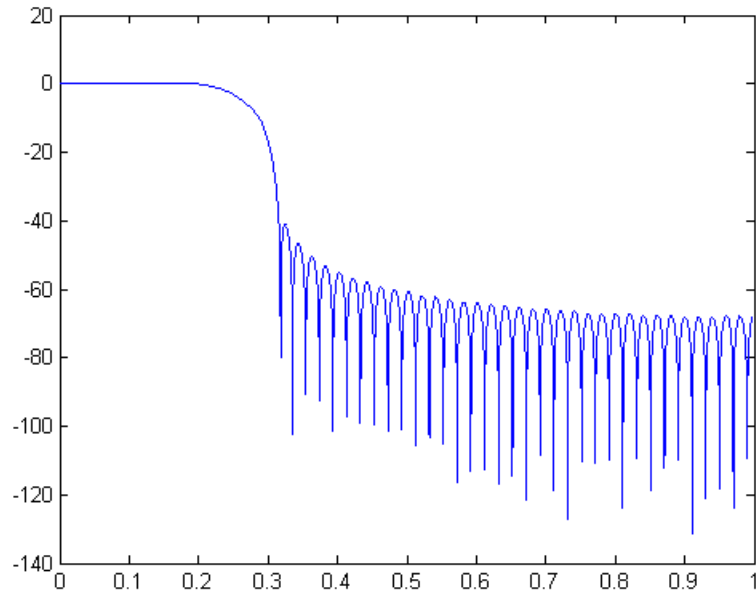
To put the input and output on the same scale, the MSBs must be weighted equally. The input MSB has a weight of 2^{11} , whereas the scaled output MSB has a weight of $2^{(29-14)} = 2^{15}$. You need to give the output MSB a weight of 2^{11} as follows:

```
yf = fi(zeros(size(yri)), true, 16, 4);
yf.bin = yri.bin;
stem(n(idx)', [xscent, 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” on page 8-89
- “Reinterpret the Output” on page 8-90

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.

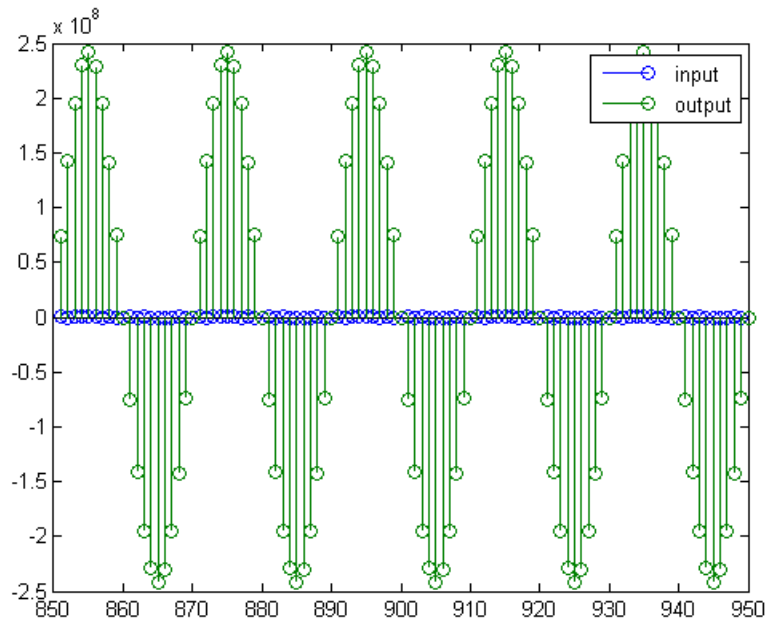
```
yint = filter(h, xsc);
norm(yd - double(yint))
```

```
ans =
```

```
0
```

You can then truncate the output to only 16 bits:

```
yout = fi(yint, true, 16);
stem(n(yidx), [xsceft, double(yout(yidx)')]);
axis([850 950 -2.5e8 2.5e8]);
legend('input', 'output');
```



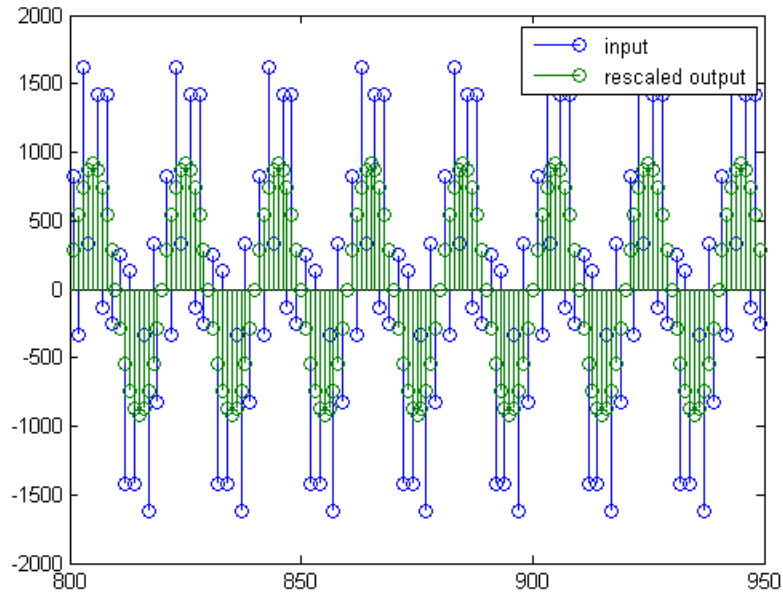
Once again, the plot shows that the input and output are at different scales. In order to scale the output so that the signals can be compared more easily in a plot, you will need to weigh the MSBs appropriately. You can compute the new fraction length using the gain of the filter when the coefficients 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)', [xscect, 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 8-92
- “Filter Implementation Blocks” on page 8-93
- “Filter Design and Implementation Blocks” on page 8-93

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

- Digital Filter
- 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, Two-Channel Synthesis Subband Filter, and Digital Filter blocks are all implementation blocks. They allow you to implement filters for which you already know the filter coefficients. The first four blocks each implement their respective filter type, while the Digital Filter block can create a variety of filter structures. All filter structures supported by the Digital Filter block support fixed-point signals.

For more information on these filter implementation blocks, see their reference pages in the Block Reference.

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.

Code Generation

Learn how to generate code for signal processing applications.

- “Understanding Code Generation” on page 9-2
- “Code Generation with System Objects” on page 9-4
- “Generate Code from MATLAB” on page 9-9
- “Generate Code from Simulink” on page 9-10

Understanding Code Generation

In this section...

“Code Generation with the Simulink® Coder™ Product” on page 9-2

“Highly Optimized Generated ANSI C Code” on page 9-3

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. For more information, see “Code Generation” in the Simulink Coder documentation.

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 Real-Time Windows (RTWIN) target.

Simulink Coder provides functions to help you set up and manage the build information for your models. For example, one of the “Build Information” 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, overwriteable arrays, and inlined algorithms provide smaller generated C code that is more efficient at run time.

Code Generation with System Objects

The following System objects support code generation in MATLAB via the `codegen` function. To use the `codegen` function, you must have a MATLAB Coder license. See “Use System Objects for Code Generation from MATLAB” for more information.

Supported DSP System Toolbox System Objects

Object	Description
Estimation	
<code>dsp.BurgAREstimator</code>	Compute estimate of autoregressive model parameters using Burg method
<code>dsp.BurgSpectrumEstimator</code>	Compute parametric spectral estimate using Burg method
<code>dsp.CepstralToLPC</code>	Convert cepstral coefficients to linear prediction coefficients
<code>dsp.LevinsonSolver</code>	Solve linear system of equations using Levinson-Durbin recursion
<code>dsp.LPCToAutocorrelation</code>	Convert linear prediction coefficients to autocorrelation coefficients
<code>dsp.LPCToCepstral</code>	Convert linear prediction coefficients to cepstral coefficients
<code>dsp.LPCToLSF</code>	Convert linear prediction coefficients to line spectral frequencies
<code>dsp.LPCToLSP</code>	Convert linear prediction coefficients to line spectral pairs
<code>dsp.LPCToRC</code>	Convert linear prediction coefficients to reflection coefficients
<code>dsp.LSFToLPC</code>	Convert line spectral frequencies to linear prediction coefficients
<code>dsp.LSPToLPC</code>	Convert line spectral pairs to linear prediction coefficients

Supported DSP System Toolbox System Objects (Continued)

Object	Description
<code>dsp.RCToAutocorrelation</code>	Convert reflection coefficients to autocorrelation coefficients
<code>dsp.RCToLPC</code>	Convert reflection coefficients to linear prediction coefficients
Filters	
<code>dsp.BiquadFilter</code>	Model biquadratic IIR (SOS) filters
<code>dsp.DigitalFilter</code>	Filter each channel of input over time using discrete-time filter implementations
<code>dsp.FIRDecimator</code>	Filter and downsample input signals
<code>dsp.FIRFilter</code>	Static or time-varying FIR filter
<code>dsp.FIRInterpolator</code>	Upsample and filter input signals
<code>dsp.FIRRateConverter</code>	Upsample, filter and downsample input signals
<code>dsp.LMSFilter</code>	Compute output, error, and weights using LMS adaptive algorithm
Math Operations	
<code>dsp.ArrayVectorAdder</code>	Add vector to array along specified dimension
<code>dsp.ArrayVectorDivider</code>	Divide array by vector along specified dimension
<code>dsp.ArrayVectorMultiplier</code>	Multiply array by vector along specified dimension
<code>dsp.ArrayVectorSubtractor</code>	Subtract vector from array along specified dimension
<code>dsp.CumulativeProduct</code>	Compute cumulative product of channel, column, or row elements
<code>dsp.CumulativeSum</code>	Compute cumulative sum of channel, column, or row elements
<code>dsp.LDLFactor</code>	Factor square Hermitian positive definite matrices into lower, upper, and diagonal components
<code>dsp.LevinsonSolver</code>	Solve linear system of equations using Levinson-Durbin recursion

Supported DSP System Toolbox System Objects (Continued)

Object	Description
<code>dsp.LowerTriangularSolver</code>	Solve $LX = B$ for X when L is lower triangular matrix
<code>dsp.LUFactor</code>	Factor square matrix into lower and upper triangular matrices
<code>dsp.Normalizer</code>	Normalize input
<code>dsp.UpperTriangularSolver</code>	Solve $UX = B$ for X when U is upper triangular matrix
Quantizers	
<code>dsp.ScalarQuantizerDecoder</code>	Convert each index value into quantized output value
<code>dsp.ScalarQuantizerEncoder</code>	Perform scalar quantization encoding
<code>dsp.VectorQuantizerDecoder</code>	Find vector quantizer codeword for given index value
<code>dsp.VectorQuantizerEncoder</code>	Perform vector quantization encoding
Signal Management	
<code>dsp.Counter</code>	Count up or down through specified range of numbers
<code>dsp.DelayLine</code>	Rebuffer sequence of inputs with one-sample shift
Signal Operations	
<code>dsp.Convolver</code>	Compute convolution of two inputs
<code>dsp.Delay</code>	Delay input by specified number of samples or frames
<code>dsp.Interpolator</code>	Interpolate values of real input samples
<code>dsp.NCO</code>	Generate real or complex sinusoidal signals
<code>dsp.PeakFinder</code>	Determine extrema (maxima or minima) in input signal
<code>dsp.PhaseUnwrapper</code>	Unwrap signal phase
<code>dsp.VariableFractionalDelay</code>	Delay input by time-varying fractional number of sample periods
<code>dsp.VariableIntegerDelay</code>	Delay input by time-varying integer number of sample periods
<code>dsp.Window</code>	Generate or apply window function

Supported DSP System Toolbox System Objects (Continued)

Object	Description
dsp.ZeroCrossingDetector	Calculate number of zero crossings of a signal
Sinks	
dsp.AudioPlayer	Write audio data to computer's audio device
dsp.AudioFileWriter	Write audio file
dsp.UDPsender	Send UDP packets to the network
Sources	
dsp.AudioFileReader	Read audio samples from an audio file
dsp.AudioRecorder	Read audio data from computer's audio device
dsp.UDPReceiver	Receive UDP packets from the network
dsp.SineWave	Generate discrete sine wave
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.RMS	Compute root-mean-square of vector elements
dsp.StandardDeviation	Compute standard deviation of vector elements
dsp.Variance	Compute variance of input or sequence of inputs
Transforms	
dsp.AnalyticSignal	Compute analytic signals of discrete-time inputs

Supported DSP System Toolbox System Objects (Continued)

Object	Description
<code>dsp.DCT</code>	Compute discrete cosine transform (DCT) of input
<code>dsp.FFT</code>	Compute fast Fourier transform (FFT) of input
<code>dsp.IDCT</code>	Compute inverse discrete cosine transform (IDCT) of input
<code>dsp.IFFT</code>	Compute inverse fast Fourier transform (IFFT) of input

Generate Code from MATLAB

Generate a C-MEX function from MATLAB code using the MATLAB Coder product

When you have a license for the MATLAB Coder product, you can generate standalone C and C++ from MATLAB code. For an example of how to generate a C-MEX function from MATLAB code, see the [Using System Objects with MATLAB Coder demo](#).

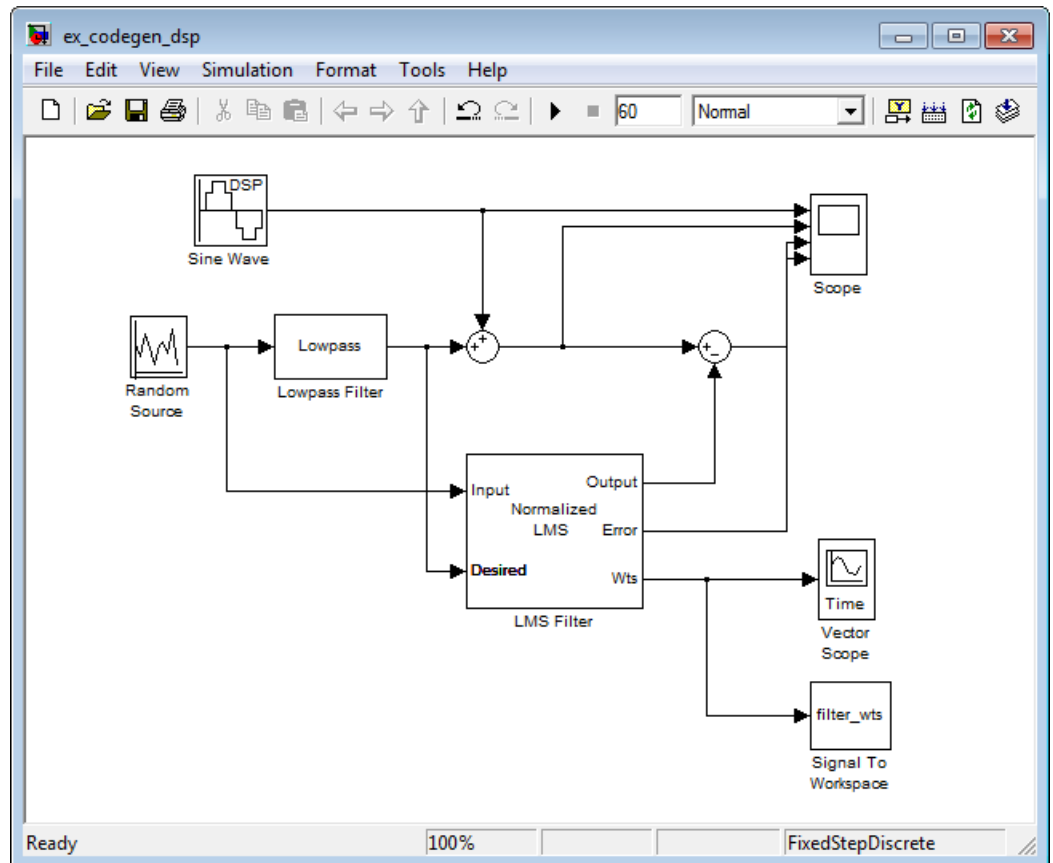
Generate Code from Simulink

In this section...
“Open and Run the Model” on page 9-10
“Generate Code from the Model” on page 9-11
“Build and Run the Generated Code” on page 9-12

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.




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 **Incremental build** 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
```

For more information about selecting a compiler, see “Selecting a Compiler on Windows Platforms” or “Selecting a Compiler on UNIX® Platforms” in the MATLAB documentation.

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 **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 **Incremental build** 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
```

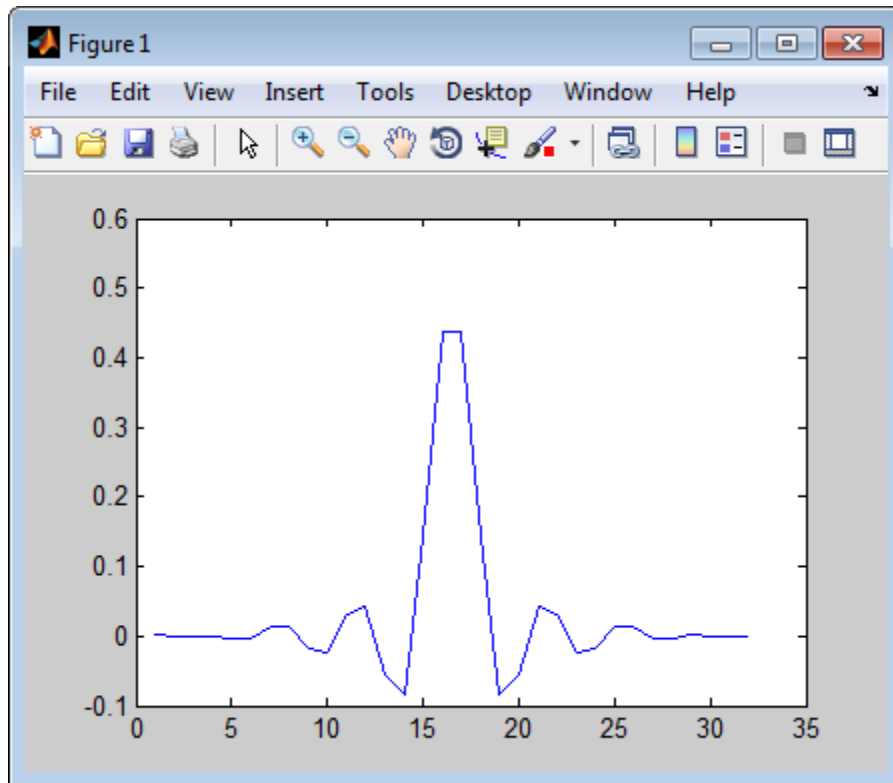
Running the generated code creates a MAT-file which contains the same variables as those generated by simulating the model. The variables in the MAT-file are named with a prefix of `rt_`. After you run the generated code, you can load the variables from the MAT-file by typing the following command at 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.

Define New System Objects

- “Define Basic System Objects” on page 10-2
- “Change Number of Step Method Inputs or Outputs” on page 10-4
- “Validate Property and Input Values” on page 10-7
- “Initialize Properties and Setup One-Time Calculations” on page 10-10
- “Set Property Values at Construction from Name-Value Pairs” on page 10-13
- “Reset Algorithm State” on page 10-16
- “Define Property Attributes” on page 10-18
- “Hide Inactive Properties” on page 10-21
- “Limit Property Values to a Finite Set of Strings” on page 10-23
- “Process Tuned Properties” on page 10-26
- “Release System Object Resources” on page 10-28
- “Define Composite System Objects” on page 10-30
- “Define Finite Source Objects” on page 10-34
- “Methods Timing” on page 10-36

Define Basic System Objects

This example shows the structure of a basic System object that increments a number by one.

The class definition file contains the minimum elements required to define a System object.

Create the Class Definition File

- 1 Create a MATLAB file named `AddOne.m` to contain the definition of your System object.

```
edit AddOne.m
```

- 2 Subclass your object from `matlab.System`. Insert this line as the first line of your file.

```
classdef AddOne < matlab.System
```

- 3 Add the `stepImpl` method, which contains the algorithm that runs when users call the `step` method on your object. You always set the `stepImpl` method access 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. You can use any name for your System object handle.

In this example, 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 from occurring.

By default, the number of inputs and outputs are both one. To change the number of inputs or outputs, use the `getNumInputsImpl` or `getNumOutputsImpl` method, respectively.

```
methods (Access=protected)
    function y = stepImpl(~, x)
        y = x + 1;
    end
end
```

Note Instead of manually creating your class definition file, you can use **File > New > System Object** to open a sample System object file in the editor. You then can edit that file, using it as guideline, to create your own System object.

Complete Class Definition File for Basic 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(~,x)
            y = x + 1;
        end
    end
end
```

See Also

[stepImpl](#) | [getNumInputsImpl](#) | [getNumOutputsImpl](#) | [matlab.System](#) |

Related Examples

- “Change Number of Step Method Inputs or Outputs” on page 10-4

More About

- “Process Data using System Objects”
- *Object-Oriented Programming*
- “Class Definition—Syntax Reference”
- “Methods — Defining Class Operations”

Change Number of Step Method Inputs or Outputs

This example shows how to specify two inputs and two outputs for the `step` method.

If you do not specify the `getNumInputsImpl` and `getNumOutputsImpl` methods, the object uses the default values of 1 input and 1 output. In this case, the user must provide an input to the `step` method.

To specify no inputs, you must explicitly set the number of inputs to 0 using the `getNumInputsImpl` method. To specify no outputs, you must explicitly return 0 in the `getNumOutputsImpl` method.

You always set the `getNumInputsImpl` and `getNumOutputsImpl` methods access to `protected` because they are internal methods that users do not directly call or run.

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 this example, 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 from occurring.

Update the Algorithm for Multiple Inputs and Outputs

Update the `stepImpl` method to accept a second input and provide a second output.

```
methods (Access=protected)
    function [y1,y2] = stepImpl(~,x1,x2)
        y1 = x1 + 1
        y2 = x2 + 1;
    end
end
```

Update the Associated Methods

Use `getNumInputsImpl` and `getNumOutputsImpl` to specify two inputs and two outputs, respectively.

```

methods (Access=protected)
    function numIn = getNumInputsImpl(~)
        numIn = 2;
    end

    function numOut = getNumOutputsImpl(~)
        numOut = 2;
    end
end

```

Complete Class Definition File with Multiple Inputs and Outputs

```

classdef AddOne < matlab.System
%ADDONE Compute output values two greater than the input values

    % All methods occur inside a methods declaration.
    % The stepImpl method has protected access
    methods(Access=protected)

        function [y1 y2] = stepImpl(~,x1,x2)
            y1 = x1 + 1;
            y2 = x2 + 1;
        end

        % getNumInputsImpl method calculates number of inputs
        function num = getNumInputsImpl(~)
            num = 2;
        end

        % getNumOutputsImpl method calculates number of outputs
        function num = getNumOutputsImpl(~)
            num = 2;
        end
    end
end

```

See Also

[getNumInputsImpl](#) | [getNumOutputsImpl](#) |

Related Examples

- “Validate Property and Input Values” on page 10-7
- “Define Basic System Objects” on page 10-2

More About

- “Class Definition—Syntax Reference”
- “Methods — Defining Class Operations”

Validate Property and Input Values

This example shows how to verify that the user's inputs and property values are valid.

You use the `validateInputsImpl` and `validatePropertiesImpl` methods to perform this validation

Note All inputs default to variable-size inputs. See “Change System Object Input Complexity or Dimensions” for more information.

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
```

```
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

```
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
        y = x + obj.Increment;
    else
        y = x + 1;
    end
end
end
end
```

See Also

[validateInputsImpl](#) | [validatePropertiesImpl](#) |

Related Examples

- “Define Basic System Objects” on page 10-2
- “Validate Property and Input Values” on page 10-7

More About

- “Methods Timing” on page 10-36
- “Property Set Methods”

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 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
```

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, data)
        obj.pFileID = fopen(obj.Filename, 'wb');
        if obj.pFileID < 0
            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,data)
            obj.pFileID = fopen(obj.Filename, 'wb');
            if obj.pFileID < 0
                error('Opening the file failed');
            end
        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

        % You indicate that no outputs are provided by returning
        % zero from getNumOutputsImpl
        function numOutputs = getNumOutputsImpl(~)
            numOutputs = 0;
        end
    end
end
```

See Also

setupImpl | releaseImpl | stepImpl |

Related Examples

- “Release System Object Resources” on page 10-28
- “Define Property Attributes” on page 10-18

More About

- “Methods Timing” on page 10-36

Set Property Values at Construction from Name-Value Pairs

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 < 0
            error('Opening the file failed');
        end
    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

    % You indicate that no outputs are provided by returning
    % zero from getNumOutputsImpl
    function numOutputs = getNumOutputsImpl(~)
        numOutputs = 0;
    end
end
end
```

See Also

narginsetProperties |

**Related
Examples**

- “Define Property Attributes” on page 10-18
- “Release System Object Resources” on page 10-28

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, which calls the resetImpl method. In this example, pCount resets to 0. See “Methods Timing” on page 10-36 for more information.

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

    % The step method takes no inputs
    function numIn = getNumInputsImpl(~)
        numIn = 0;
    end
end
end
end
```

See Also [resetImpl](#) |

**More
About**

- “Methods Timing” on page 10-36

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

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, but the value displays as `true` or `false`, respectively. 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 a tunable property.

```
properties (Logical)
    Increment = true;
end
```

Specify Property as Positive Integer

In this example, the private property `pCount` is constrained to accept only real, positive integers. You cannot use sparse values.

```
properties (PositiveInteger)
    Count
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
- Does not have any of these attributes: `Nontunable`, `Dependent`, `Abstract`, 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 starting at an initial value

    % These properties are nontunable. They cannot be changed
    % after the setup or step method has been called.
    properties (Nontunable)
        % The initial value of the counter
        InitialValue = 0
    end

    properties (Logical)
        % Whether to increment the counter
        Increment = true
    end
end
```

```
end

% Count state variable
properties (DiscreteState, PositiveInteger)
    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.Count + 1;
        end
        c = obj.Count;
    end
    % Setup the Count state variable
    function setupImpl(obj)
        obj.Count = 0;
    end
    % Reset the counter to zero.
    function resetImpl(obj)
        obj.Count = obj.InitialValue;
    end
    % The step method takes no inputs
    function numIn = getNumInputsImpl(~)
        numIn = 0;
    end
end
end
```

More About

- “Class Attributes”
- “What are System Object Locking and Property Tunability?”
- “Methods Timing” on page 10-36

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.

```
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

```
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;
    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

% The step method takes no inputs
function numIn = getNumInputsImpl(~)
    numIn = 0;
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 a Finite Set of Strings

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
function n = getNumInputsImpl(~)
    n = 0;
end
function n = getNumOutputsImpl(~)
    n = 0;
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` |

More About

- “Enumerations”

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)
        obj.pLookupTable = obj.MiddleC * ...
            (1+log(1:obj.NumNotes)/log(12));
    end
end
```

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;
        end
    end
end
```

```
        obj.pLookupTable = obj.MiddleC * ...
            (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)
        % Generate a lookup table of note frequencies
        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 illustrates the use of StringSets
    %
    properties
        Color = 'blue'
    end

    properties (Hidden)
        % 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
    end
end
```

```
end

function releaseImpl(obj)
    cla(Whiteboard.getWhiteboard());
    hold('on');
end

function n = getNumInputsImpl(~)
    n = 0;
end
function n = getNumOutputsImpl(~)
    n = 0;
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 [isInactivePropertyImpl](#) |

Related Examples

- “Initialize Properties and Setup One-Time Calculations” on page 10-10

Define Composite System Objects

This example shows how to define System objects that include other System objects.

This example defines a filter System object from an FIR System object and an IIR System object.

Store System Objects in Properties

To define a System object from other System objects, store those objects in your class definition file as properties. In this example, FIR and IIR are separate System objects defined in their own class-definition files. You use those two objects to calculate the pFir and pIir property values.

```
properties (Nontunable, Access = private)
    pFir % store the FIR filter
    pIir % store the IIR filter
end

methods
    function obj = Filter(varargin)
        setProperties(obj, nargin, varargin{:});
        obj.pFir = FIR(obj.zero);
        obj.pIir = IIR(obj.pole);
    end
end
```

Complete Class Definition File of Composite System Object

```
classdef Filter < matlab.System
    %Filter System object with a single pole and a single zero
    %
    % This System object illustrates composition by
    % composing an instance of itself.
    %

    properties (Nontunable)
        zero = 0.01
        pole = 0.5
    end
end
```

```
end

properties (Nontunable,Access=private)
    pZero % store the FIR filter
    pPole % store the IIR filter
end

methods
    function obj = Filter(varargin)
        setProperties(obj,nargin, varargin{:});
        % Create instances of FIR and IIR as
        % private properties
        obj.pZero = Zero(obj.zero);
        obj.pPole = Pole(obj.pole);
    end
end

methods (Access=protected)
    function setupImpl(obj,x)
        setup(obj.pZero,x);
        setup(obj.pPole,x);
    end

    function resetImpl(obj)
        reset(obj.pZero);
        reset(obj.pPole);
    end

    function y = stepImpl(obj,x)
        y = step(obj.pZero,x) + step(obj.pPole,x);
    end

    function releaseImpl(obj)
        release(obj.pZero);
        release(obj.pPole);
    end
end
end
```

Class Definition File for FIR Component of Filter

```
classdef Pole < matlab.System

    properties
        Den = 1
    end

    properties (Access=private)
        tap = 0
    end

    methods
        function obj = Pole(varargin)
            setProperties(obj,nargin,varargin{:},'Den');
        end
    end

    methods (Access=protected)
        function y = stepImpl(obj,x)
            y = x + obj.tap * obj.Den;
            obj.tap = y;
        end
    end

end
```

Class Definition File for IIR Component of Filter

```
classdef Zero < matlab.System

    properties
        Num = 1
    end

    properties (Access=private)
        tap = 0
    end

    methods
        function obj = Zero(varargin)
            setProperties(obj, nargin,varargin{:},'Num');
        end
    end

end
```



```
        end
    end

    methods (Access=protected)
        function y = stepImpl(obj,x)
            y = x + obj.tap * obj.Num;
            obj.tap = x;
        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
```

```
        out = 0;
    end
end

function bDone = isDoneImpl(obj)
    bDone = obj.NumSteps==2;
end
end

methods (Access=protected)
    function n = getNumInputsImpl(~)
        n = 0;
    end
    function n = getNumOutputsImpl(~)
        n = 1;
    end
end
end

end
```

See Also [matlab.system.mixin.FiniteSource](#) |

More About

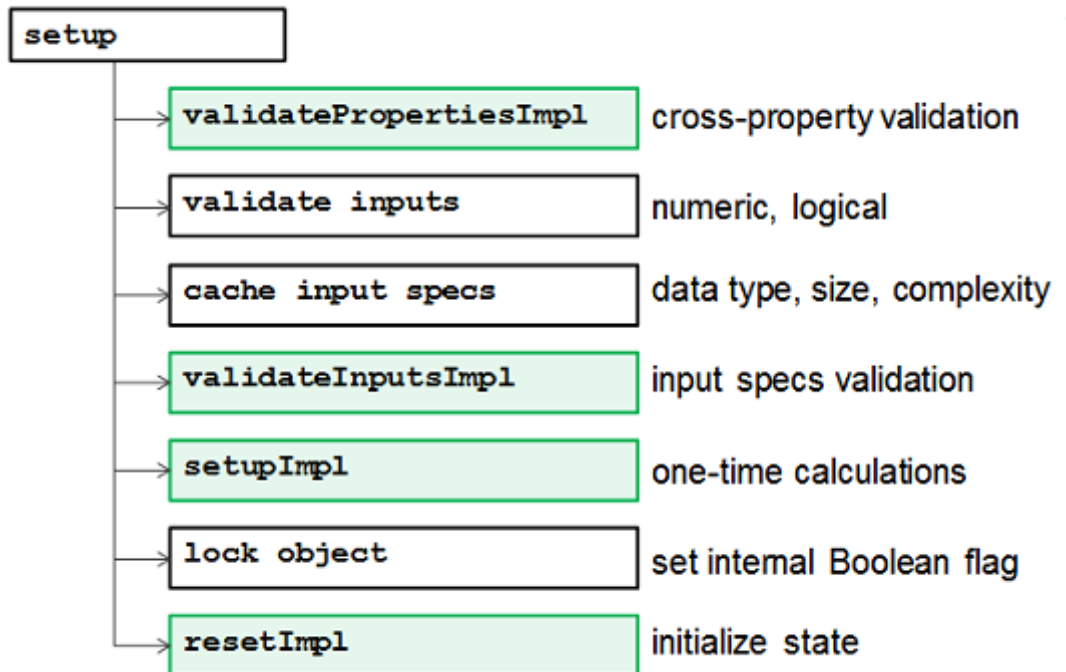
- “Subclassing Multiple Classes”

Methods Timing

In this section...
“Setup Method Call Sequence” on page 10-36
“Step Method Call Sequence” on page 10-37
“Reset Method Call Sequence” on page 10-37
“Release Method Call Sequence” on page 10-38

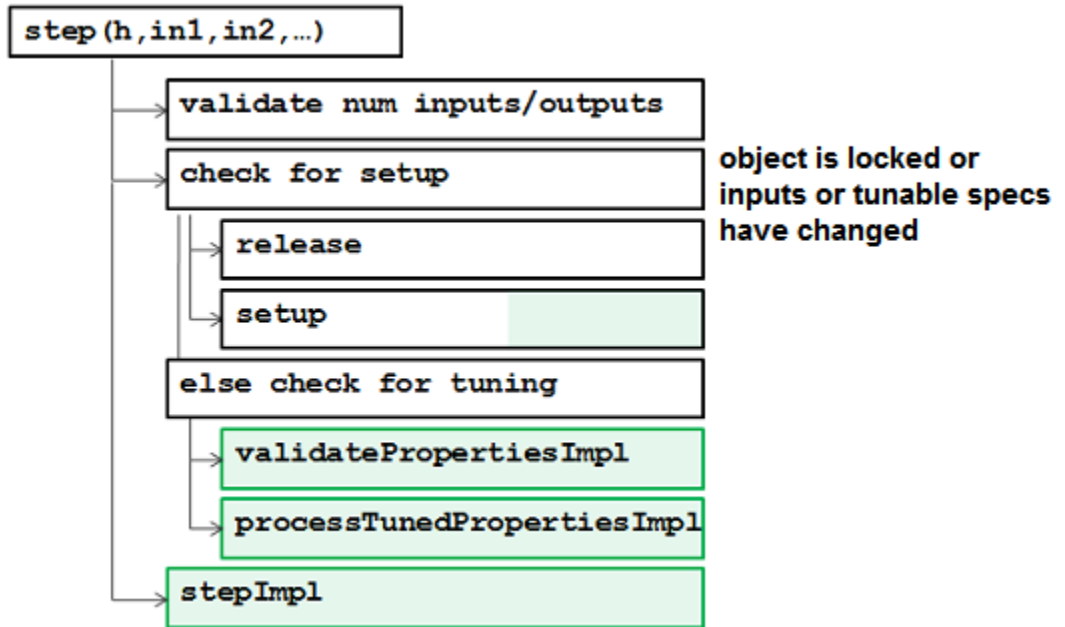
Setup Method Call Sequence

This hierarchy shows the actions performed when you call the setup method.



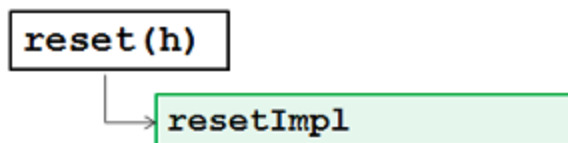
Step Method Call Sequence

This hierarchy shows the actions performed when you call the step method.



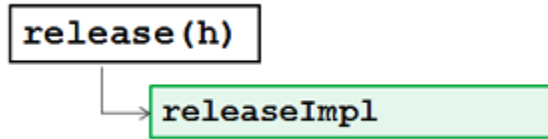
Reset Method Call Sequence

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

setupImpl | stepImpl | releaseImpl | resetImpl |

Related Examples

- “Release System Object Resources” on page 10-28
- “Reset Algorithm State” on page 10-16
- “Set Property Values at Construction from Name-Value Pairs” on page 10-13
- “Define Basic System Objects” on page 10-2

More About

- “What are System Object Methods?”
- “The Step Method”
- “Common Methods”

Bibliography

- “Advanced Filters” on page A-2
- “Adaptive Filters” on page A-3
- “Multirate Filters” on page A-4
- “Frequency Transformations” on page A-5
- “Fixed-Point Filters” on page A-6

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.

Adaptive Filters

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

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

Multirate Filters

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

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

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

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

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

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

Frequency Transformations

- [1] Constantinides, A.G., "Spectral Transformations for Digital Filters," *IEEE Proceedings*, Vol. 117, No. 8, pp. 1585-1590, August 1970.

- [2] Nowrouzian, B., and A.G. Constantinides, "Prototype Reference Transfer Function Parameters in the Discrete-Time Frequency Transformations," *Proceedings 33rd Midwest Symposium on Circuits and Systems*, Calgary, Canada, Vol. 2, pp. 1078-1082, August 1990.

- [3] Feyh, G., J.C. Franchitti, and C.T. Mullis, "Allpass Filter Interpolation and Frequency Transformation Problem," *Proceedings 20th Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, California, pp. 164-168, November 1986.

- [4] Krukowski, A., G.D. Cain, and I. Kale, "Custom Designed High-Order Frequency Transformations for IIR Filters," *38th Midwest Symposium on Circuits and Systems (MWSCAS'95)*, Rio de Janeiro, Brazil, August 1995.

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*.

A

- adaptfilt object
 - apply to data 4-13
- adaptive filter object 4-13
 - See also* adaptfilt object
- adaptive filters 4-51
 - creating 4-53
 - customizing 4-58
- add
 - samples 2-31
- algebraic loop errors 2-64
- algorithmic delay 2-57
 - adjustable 2-60
 - and initial conditions 2-60
 - basic 2-60
 - excess 2-63
 - relation to latency 2-63
 - zero 2-57
- analog filter designs 3-159
 - See also* filter designs
- angular frequency 1-2
 - See also* periods
- arithmetic operations
 - fixed-point 8-10
- arrays
 - importing 1-55
- attenuation
 - stopband 3-159
- auto-promoting rates 1-9
- avoiding unintended rate conversion 2-24

B

- band configurations 3-159
- bandpass filter designs
 - analog, available parameters 3-159
- bandstop filter designs
 - analog, available parameters 3-159
- basic
 - statistical operations 7-3

- basic algorithmic delay 2-60
- benefits
 - frame-based processing 2-56
- Bit-reversed order 6-12
- block parameters
 - fixed-point 8-28
- block rate types 2-64
- blocks
 - multirate 2-64
 - single-rate 2-64
- Buffer overlap parameter
 - negative values for 2-42
- buffering 2-31
 - altering the sample period of the signal 2-36
 - altering the signal 2-32
 - causing unintentional rate conversions 2-30
 - frame-based signals into other frame-based signals 2-47
 - internal 2-42
 - preserving the sample period of the signal 2-33
 - sample-based signals into frame-based signals 2-39
 - sample-based signals into frame-based signals with overlap 2-42
- butter function 3-160
- Butterworth filter designs
 - analog 3-159
 - band configurations for 3-159

C

- C code
 - optimization 9-3
- casts
 - fixed-point 8-16
- changing
 - frame sizes 2-21
 - the frame size of a signal 2-33

- changing quantized filter properties in
 - FDATool 3-30
 - cheby1 function 3-160
 - cheby2 function 3-160
 - Chebyshev type I filter designs
 - analog 3-159
 - band configurations for 3-159
 - Chebyshev type II filter designs
 - analog 3-159
 - band configurations for 3-159
 - choosing
 - filter design blocks 3-123
 - code
 - generating 9-10
 - code generation 9-10
 - fixed-point 8-3
 - generic real-time (GRT) 2-57
 - optimization 9-3
 - overview 9-2
 - signal processing objects 9-4
 - understanding 9-2
 - with Simulink Coder 9-2
 - coefficients
 - exporting 3-82
 - combining
 - frame-based signals 1-33
 - multichannel sample-based signals 1-29
 - single-channel sample-based signals 1-26
 - complex multiplication
 - fixed-point 8-13
 - computational delay 2-55
 - reducing 2-56
 - concatenating
 - frame-based signals 1-33
 - multichannel sample-based signals 1-29
 - single-channel sample-based signals 1-26
 - concepts
 - frame rate 2-8
 - sample rate 2-8
 - configuring
 - vector quantization model 8-60
 - continuous-time
 - discretizing signals 1-11
 - signals 1-11
 - source blocks 1-11
 - controls
 - FDATool 3-21
 - conventions
 - time and frequency 1-2
 - converting 2-18
 - frame rates 2-18
 - frame-based signals into other frame-based signals 2-47
 - sample-based signals into frame-based signals 2-39
 - sample-based signals into frame-based signals with overlap 2-42
 - See also* rate conversion
 - converting filter structures in FDATool 3-35
 - creating
 - 1-D vector signal 1-14
 - adaptive filters 4-53
 - fixed-point filters 3-134
 - frame-based signals 1-19
 - multichannel frame-based signals 1-32
 - multichannel sample-based signals 1-26
 - sample-based signals 1-13
 - scalar quantizers 8-54
 - vector quantizers 8-58
 - creation of
 - spectrograms 6-34
 - customizing
 - adaptive filters 4-58
- ## D
- data types 3-161 8-65
 - fixed 3-162 8-66
 - fixed-point
 - floating-point 3-161 8-65

- single 3-162 8-66
- decimation factor 5-2
- decimator 5-2
- deconstructing
 - multichannel frame-based signals 1-43
 - multichannel frame-based signals into individual signals 1-43
 - multichannel sample-based signals 1-36
 - multichannel sample-based signals into individual signals 1-36
 - multichannel sample-based signals into other multichannel signals 1-39
- delay
 - algorithmic 2-57
 - computational 2-55
 - rebuffering 2-50
 - relation to latency 2-63
- delete
 - samples 2-31
- demos
 - multirate filtering 5-21
- design methods 3-6
 - customize 3-8
- designing
 - adaptive filters 4-53
 - fixed-point filters 3-134
 - scalar quantizers 8-54
 - vector quantizers 8-58
- designing fixed-point multirate filters 3-80
- designing multirate filters 3-80
- Digital Filter block
 - filtering noise with 3-150
- Digital Filter Design block
 - filtering noise with 3-128
- digital frequency 1-2
 - defined 1-2
 - See also* periods
- discrete-time signals 1-2
 - characteristics 1-2
 - defined 1-2

- terminology 1-2
- See also* signals
- discretizing a continuous-time signal 1-11
- displaying
 - frequency-domain data 1-90
 - line widths 2-18
 - power spectrum of speech signal 6-31
 - spectrograms 6-40
 - time-domain data 1-73
- downsampling 2-18
- See also* rate conversion

E

- ellip function 3-160
- elliptic filter designs
 - analog 3-159
 - band configurations for 3-159
- errors
 - algebraic loop 2-64
 - due to continuous-time input to a discrete-time block 1-11
 - sample-rate mismatch 1-6
- estimation
 - power spectrum 6-23
 - example 6-24
- examples
 - latency 2-65
 - multirate filtering 5-21
- exporting
 - frame-based signals 1-67
 - sample-based signals 1-59
- exporting individual phase coefficients of a polyphase filter 3-82
- exporting quantized filters in FDATool 3-56

F

- factoring matrices 7-8
- FDATool

- about importing and exporting filters 3-53
 - about quantization mode 3-20
 - apply option 3-22
 - changing quantized filter properties 3-30
 - controls 3-21
 - convert structure option 3-35
 - converting filter structures 3-35
 - exporting quantized filters 3-56
 - frequency point to transform 3-65
 - import filter dialog box 3-55
 - importable filter structures 3-53
 - importing filters 3-54
 - original filter type 3-61
 - quantized filter properties 3-22
 - quantizing filters 3-22
 - quantizing reference filters 3-29
 - set quantization parameters dialog 3-22
 - setting properties 3-22
 - specify desired frequency location 3-66
 - switching to quantization mode 3-20
 - transform filters in FDATool 3-66
 - transformed filter type 3-65
 - user options 3-21
- FFT block
- using 6-2
- FFT length parameter 2-27
- filter algorithm 3-6
- choosing 3-6
- filter band configurations 3-159
- filter cost 5-2
- filter data 3-10
- filter design
- adaptive filter 4-42
 - customize algorithm 3-8
 - filter analysis 3-9
 - Filter Object 3-8
 - flow chart
 - flow diagram 3-2
 - multirate filters in FDATool 3-69
 - process 3-2
 - single-rate filters in FDATool 3-16
 - specification 3-4
 - Specifications Object 3-4
- Filter Design
- Multirate 5-8
 - Multistage 5-8
 - Narrow Transition-Band 5-8
- filter design blocks
- choosing 3-123
- filter design parameters 3-4
- filter designs
- available parameters 3-159
 - butter function 3-160
 - Butterworth 3-159
 - cheby1 function 3-160
 - cheby2 function 3-160
 - Chebyshev type I 3-159
 - Chebyshev type II 3-159
 - continuous-time 3-159
 - ellip function 3-160
 - elliptic 3-159
 - passband ripple 3-159
 - stopband attenuation 3-159
- filter response 3-4
- filters
- adaptive 4-51
 - creating a highpass filter 3-126
 - creating a lowpass filter 3-125
 - exporting as MAT-file 3-58
 - exporting as text file 3-57
 - exporting from FDATool 3-56
 - Filter Realization Wizard 3-134
 - filtering noise with Digital Filter
 - blocks 3-150
 - filtering noise with Digital Filter Design
 - blocks 3-128
 - fixed-point 3-188 8-92
 - getting filter coefficients after exporting 3-57
 - implementing a highpass filter 3-149
 - implementing a lowpass filter 3-148

- importing and exporting 3-53
- importing into FDATool 3-54
- filters, export as MAT-file 3-58
- fixed point
 - System object preferences 8-25
- fixed point properties
 - System objects 8-26
- fixed-point attributes, specification
 - at the block level 8-28
 - at the system level 8-31
- fixed-point block parameters
 - setting 8-28
- fixed-point code generation 8-3
- fixed-point data types 8-4
 - addition 8-12
 - arithmetic operations 8-10
 - attributes 8-28
 - casts 8-16
 - complex multiplication 8-13
 - concepts 8-4
 - filters 3-188 8-92
 - logging 8-31
 - modular arithmetic 8-10
 - multiplication 8-13
 - overflow handling 8-6
 - precision 8-6
 - range 8-6
 - rounding 8-7
 - saturation 8-6
 - scaling 8-5
 - subtraction 8-12
 - terminology 8-4
 - two's complement 8-11
 - wrapping 8-6
- fixed-point development 8-2
- fixed-point filter 3-161 8-65
 - conversion from floating-point 3-163 8-67
 - definition 3-161 8-65
- fixed-point filters
 - designing and implementing 3-134
 - fixed-point multirate filters 3-80
- Fixed-Point Tool 8-31
- fixed-step solvers 1-6
- frame periods 2-17
 - altered by unbuffering 2-51
 - converting 2-17
 - multiple 2-18
 - related to sample period and frame size 2-8
 - Simulink Probe block 2-11
 - See also* rate conversion
- frame rates 1-9
 - auto-promoting 1-9
 - color coding 2-15
 - concepts 2-8
 - inspecting 2-15
 - See also* frame periods
- frame rebuffering
 - blocks for 2-30
- frame sizes 2-17
 - changing 2-33
 - converting 2-17
 - converting by rebuffering 2-17
 - direct rate conversion 2-17
 - maintaining a constant sample rate 2-31
 - related to sample period and frame period 2-8
 - See also* rate conversion
- frame-based
 - processing 2-5
- frame-based processing
 - benefits 2-56
 - latency 2-7
- frame-based signals
 - benefits of 2-6
 - combining 1-33
 - concatenating 1-33
 - converting to other frame-based signals 2-47
 - creating 1-19
 - deconstructing multichannel signals 1-43
 - exporting 1-67

- importing 1-64
- importing and exporting 1-64
- reordering channels in a multichannel signal 1-48
- separating multichannel signals 1-43
- unbuffering to sample-based signals 2-51

frame-rate adjustment

- rate conversion 2-18

frame-size adjustment

- rate conversion 2-21

frequencies 1-2

- normalized 3-159
- normalized linear 1-2
- terminology 1-2
- See also* periods

frequency point to transform 3-65

frequency-domain data

- displaying 1-90
- transforming it into the time domain 6-7

function for opening FDATool 3-20

function reuse 9-3

G

- generated code
 - generic real-time (GRT) 2-57
- getting filter coefficients after exporting 3-57

H

- highpass filter designs
 - continuous-time 3-159
- Hz (hertz) 1-2
 - defined 1-2
 - See also* sample periods

I

- IFFT block
 - using 6-7
- import filter dialog box in FDATool 3-55

- import filter dialog box options 3-55
 - discrete-time filter 3-55
 - frequency units 3-55
- import/export filters in FDATool 3-53
- importing
 - arrays 1-55
 - frame-based signals 1-64
 - pages of an array 1-55
 - sample-based matrices 1-55
 - sample-based signals 1-52
 - sample-based vector signals 1-52
- importing and exporting
 - frame-based signals 1-64
 - sample-based signals 1-52
- importing filters 3-54
- importing quantized filters in FDATool 3-54
- inherit via internal rule 8-32
- inheriting sample periods 1-11
- initial conditions
 - with basic algorithmic delay 2-60
- input frame periods
 - defined 2-8
- inspecting
 - frame periods 2-11
 - frame rates 2-15
 - sample periods 2-10
 - sample rates 2-13
- interpolator 5-2
- inversion of matrices 7-9

L

- latency 2-63
 - due to frame-based processing 2-7
 - predicting 2-65
 - reducing 2-63
 - relation to delay 2-63
- libraries
 - Statistics 7-2
- line widths

- displaying 2-18
- linear algebra
 - solving linear systems 7-6
- Linear order 6-12
- logging
 - fixed-point data types 8-31
- lowpass filter designs
 - continuous-time 3-159

M

- M factor 5-2
- matrices
 - factoring 7-8
 - inverting 7-9
- maximum 7-2
- mean 7-2
- minimum 7-2
- models
 - multirate 2-18
- modes
 - tasking 2-63
- modular arithmetic 8-10
- multichannel
 - frame-based signals 1-32
 - sample-based signals 1-26
- multiplication
 - fixed-point 8-13
- multirate
 - blocks 2-64
 - demos 5-21
 - examples 5-21
 - models 2-64
- multirate filter
 - definition 5-2
- multirate filters
 - designing 3-80
- multistage filter
 - definition 5-6
 - uses 5-6

- multitasking mode 2-63

N

- normalized frequencies 1-2
 - defined 1-2
 - See also* frequencies
- Nyquist frequency
 - defined 1-2
- Nyquist rate 1-2

O

- opening FDATool
 - function for 3-20
- optimization
 - code generation 9-3
- options
 - FDATool 3-21
- original filter type 3-61
- output frame periods
 - defined 2-8
- overflow handling 8-6
- overlapping buffers
 - causing unintentional rate conversions 2-30

P

- pages of an array
 - importing 1-55
- parameter reuse 9-3
- parameters
 - Buffer overlap, negative values for 2-42
 - continuous-time filter 3-159
 - FFT length 2-27
 - normalized frequency 3-159
- Partial Unbuffer block 2-32
- partial unbuffering 2-31
- passband ripple
 - analog filter 3-159
- performance

- improving 2-6
- periods 1-2
 - defined 1-2
 - See also* sample periods *and* frame periods
- power spectrum
 - estimation 6-24
 - of speech signal 6-24
 - viewing 6-31
- power spectrum estimation 6-23
- precision
 - fixed-point data types 8-6
- predicting
 - tasking latency 2-65
- preferences 8-25
- preventing unintended rate conversion 2-24
- Probe block 2-10

Q

- quantization mode in FDATool 3-20
- quantized filter properties
 - changing in FDATool 3-30
- quantizers
 - scalar 8-51
 - vector 8-58
- quantizing filters in FDATool 3-29

R

- range
 - fixed-point data types 8-6
- rate conversion
 - avoiding 2-24
 - avoiding rate-mismatch errors 1-7
 - blocks for 2-17
 - by unbuffering 2-51
 - direct 2-17
 - frame-rate adjustment 2-18
 - frame-size adjustment 2-21
- rate types

- block 2-64
- model 2-64
- rates 2-8
 - auto-promoting 1-9
 - See also* sample periods *and* frame periods
- realize data 3-10
- rebuffering 2-31
 - altering the sample period of the signal 2-36
 - altering the signal 2-32
 - causing unintentional rate conversions 2-30
 - delay 2-50
 - preserving the sample period of the signal 2-33
- reducing
 - latency 2-63
- reflection coefficients
 - identifying 8-53
- reordering channels
 - in multichannel frame-based signals 1-48
- residual signal
 - identifying 8-53
- reuse of
 - functions 9-3
 - parameters 9-3
- ripple
 - passband 3-159
- rounding
 - fixed-point data types 8-7
- running
 - vector quantization model 8-60
- running operations 7-4

S

- sample frequency 1-2
 - definition 1-2
 - See also* sample periods
- sample modes 2-65
- sample periods 1-2
 - altered by unbuffering 2-51

- Buffer block 2-32
- continuous-time 1-11
- defined 1-2
- for frame-based signals 2-8
- inherited 1-11
- maintaining constant 2-31
- nonsource blocks 1-11
- of source blocks 1-11
- Rebuffer block 2-32
- related to frame period and frame size 2-8
- Simulink Probe block 2-10
- See also* frame periods *and* sample times
- sample rates 1-2
 - auto-promoting 1-9
 - color coding 2-13
 - concepts 2-8
 - defined 1-2
 - inspecting 2-13
 - See also* sample periods
- sample time
 - of original time series parameter 2-30
- sample times 1-2
 - defined 1-2
 - in DSP System Toolbox 1-3
 - shifting with sample-time offsets 2-10
 - See also* sample periods *and* frame periods
- sample-based
 - processing 2-4
- sample-based signals
 - combining multichannel signals 1-29
 - combining single-channel signals 1-26
 - concatenating multichannel signals 1-29
 - concatenating single-channel signals 1-26
 - converting to frame-based 2-39
 - converting to frame-based with overlap 2-42
 - creating 1-13
 - deconstructing multichannel signals 1-36
 - exporting 1-59
 - importing 1-52
 - importing and exporting 1-52
 - multichannel 1-26
 - splitting multichannel signals 1-36
- samples
 - adding 2-31
 - deleting 2-31
 - rearranging 2-32
- sampling 2-8
 - See also* sample periods *and* frame periods
- saturation 8-6
- scalar quantizers 8-51
 - creating 8-54
- scaling 8-5
- separating
 - multichannel frame-based signals 1-43
- sequences
 - defining a discrete-time signal 1-2
- set quantization parameters dialog 3-22
- setting filter properties in FDATool 3-22
- signals
 - benefits of frame-based 2-6
 - characteristics 1-2
 - continuous-time 1-11
 - converting frame-based to sample-based 2-51
 - definition of discrete-time 1-2
 - definition of frequency 1-2
 - discrete-time terminology 1-2
 - inspecting the frame period of 2-11
 - inspecting the sample period of 2-10
 - Nyquist frequency 1-2
 - Nyquist rate 1-2
 - terminology 1-3
- simulations
 - running from the command line 2-56
- Simulink Coder
 - code generation 9-2
 - generating code 9-10
 - generating generic real-time (GRT) code 2-57
- single-rate
 - blocks 2-64
 - models 2-64

- single-tasking mode 2-63
 - size of a frame 2-17
 - sliding windows
 - example 7-3
 - solvers
 - fixed-step 1-6
 - variable-step 1-6
 - solving
 - linear systems 7-6
 - source blocks
 - defined 1-11
 - sample periods of 1-11
 - sources
 - sample periods of 1-11
 - specifying desired frequency location 3-66
 - spectrogram
 - creating 6-34
 - of speech signal 6-34
 - viewing 6-40
 - speech
 - analysis and synthesis 8-51
 - splitting
 - multichannel frame-based signals into
 - individual signals 1-43
 - multichannel sample-based signals 1-36
 - multichannel sample-based signals into
 - individual signals 1-36
 - multichannel sample-based signals into
 - other multichannel signals 1-39
 - standard deviation 7-2
 - starting FDATool 3-20
 - statistics
 - operations 7-2
 - Statistics library 7-2
 - stopband attenuation 3-159
 - symbols
 - time and frequency 1-2
 - System object
 - fixed point 8-21
 - preferences 8-25
 - setting frame-based processing 2-4
 - system-level settings
 - fixed-point 8-31
- ## T
- tasking latency 2-63
 - example 2-65
 - predicting 2-65
 - tasking modes 2-63
 - terminology
 - sample time and sample period 1-3
 - time and frequency 1-2
 - throughput rates
 - increasing 2-7
 - time-domain data
 - displaying 1-73
 - transforming it into the frequency domain 6-2
 - transform filter
 - frequency point to transform 3-65
 - original filter type 3-61
 - specify desired frequency location 3-66
 - transformed filter type 3-65
 - transformed filter type 3-65
 - transforming
 - frequency-domain data 6-7
 - time-domain data 6-2
 - two's complement 8-11
- ## U
- unbuffering 2-51
 - and rate conversion 2-51
 - partial 2-31
 - to a sample-based signal 2-32
 - units of time and frequency measures 1-2
 - upsampling 2-18
 - See also* rate conversion
 - using

- the FFT block 6-2
- the IFFT block 6-7
- using `adaptfilt` objects 4-13
- using `FDATool` 3-54

V

- variable-step solver 1-6
- vector quantizers 8-58
 - configuring the model 8-60
 - creating 8-58
 - running the model 8-60
- viewing
 - frequency-domain data 1-90
 - power spectrum of speech signal 6-31
 - spectrogram of speech signal 6-40
 - time-domain data 1-73

W

- Wavelet reconstruction
 - channel latency 6-14
- wrapping
 - fixed-point data types 8-6

Z

- zero algorithmic delay 2-57
- Zero-Order Hold block 1-11
- zero-padding 2-27
 - causing unintentional rate conversions 2-30
- zeros
 - padding with 2-32