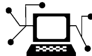
Signal Processing Blockset™ 7

User's Guide

The MathWorks™
Accelerating the pace of engineering and science

How to Contact The MathWorks



www.mathworks.com	Web
comp.soft-sys.matlab	Newsgroup
www.mathworks.com/contact_TS.html	Technical Support
@ suggest@mathworks.com	Product enhancement suggestions
bugs@mathworks.com	Bug reports
doc@mathworks.com	Documentation error reports
service@mathworks.com	Order status, license renewals, passcodes
info@mathworks.com	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

Signal Processing Blockset™ User's Guide

© COPYRIGHT 1995–2010 The 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

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

Revision History

April 1995	First printing	New for Version 1.0
May 1997	Second printing	Revised for Version 2.0
January 1998	Third printing	Revised for Version 2.2 (Release 10)
January 1999	Fourth printing	Revised for Version 3.0 (Release 11)
November 2000	Fifth printing	Revised for Version 4.0 (Release 12)
June 2001	Online only	Revised for Version 4.1 (Release 12.1)
July 2002	Sixth printing	Revised for Version 5.0 (Release 13)
April 2003	Seventh printing	Revised for Version 5.1 (Release 13SP1)
June 2004	Online only	Revised for Version 6.0 (Release 14) (Renamed from DSP Blockset User's Guide)
October 2004	Online only	Revised for Version 6.0.1 (Release 14SP1)
March 2005	Online only	Revised for Version 6.1 (Release 14SP2)
September 2005	Online only	Revised for Version 6.2 (Release 14SP3)
March 2006	Online only	Revised for Version 6.3 (Release 2006a)
September 2006	Online only	Revised for Version 6.4 (Release 2006b)
March 2007	Online only	Revised for Version 6.5 (Release 2007a)
September 2007	Online only	Revised for Version 6.6 (Release 2007b)
March 2008	Online only	Revised for Version 6.7 (Release 2008a)
October 2008	Online only	Revised for Version 6.8 (Release 2008b)
March 2009	Online only	Revised for Version 6.9 (Release 2009a)
September 2009	Online only	Revised for Version 6.10 (Release 2009b)
March 2010	Online only	Revised for Version 7.0 (Release 2010a)

Contents

1 Working with Signals

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**

Sample-Based Signals **1-13**
Sample-Based Single Channel Signals **1-13**
Sample-Based Multichannel Signals **1-13**

Frame-Based Signals **1-15**
Frame-Based Single Channel Signals **1-15**
Frame-Based Multichannel Signals **1-15**
Benefits of Frame-Based Processing **1-16**

Creating Sample-Based Signals **1-19**
Using the Constant Block **1-19**
Using the Signal from Workspace Block **1-21**

Creating Frame-Based Signals **1-25**
Using the Sine Wave Block **1-25**
Using the Signal from Workspace Block **1-28**

Creating Multichannel Sample-Based Signals **1-32**
Multichannel Sample-Based Signals **1-32**
Combining Single-Channel Sample-Based Signals **1-32**
Combining Multichannel Sample-Based Signals **1-35**

Creating Multichannel Frame-Based Signals **1-38**

Multichannel Frame-Based Signals **1-38**
Combining Frame-Based Signals **1-39**

Deconstructing Multichannel Sample-Based Signals .. **1-42**
Splitting Multichannel Sample-Based Signals into Individual Signals **1-42**
Splitting Multichannel Sample-Based Signals into Several Multichannel Signals **1-45**

Deconstructing Multichannel Frame-Based Signals ... **1-49**
Splitting Multichannel Frame-Based Signals into Individual Signals **1-49**
Reordering Channels in Multichannel Frame-Based Signals **1-54**

Importing and Exporting Sample-Based Signals **1-58**
Importing Sample-Based Vector Signals **1-58**
Importing Sample-Based Matrix Signals **1-61**
Exporting Sample-Based Signals **1-65**

Importing and Exporting Frame-Based Signals **1-70**
Importing Frame-Based Signals **1-70**
Exporting Frame-Based Signals **1-73**

Displaying Time-Domain Data **1-79**
Displaying Time Domain Data in the Vector Scope **1-79**
Displaying Time-Domain Data in the Time Scope **1-82**

Displaying Frequency-Domain Data **1-100**

2 Advanced Signal Concepts

Inspecting Sample Rates and Frame Rates **2-2**
Sample Rate and Frame Rate Concepts **2-2**
Inspecting Sample-Based Signals Using the Probe Block .. **2-3**
Inspecting Frame-Based Signals Using the Probe Block .. **2-5**
Inspecting Sample-Based Signals Using Color Coding **2-7**

Inspecting Frame-Based Signals Using Color Coding **2-9**

Converting Sample and Frame Rates **2-11**
Rate Conversion Blocks **2-11**
Rate Conversion by Frame-Rate Adjustment **2-12**
Rate Conversion by Frame-Size Adjustment **2-15**
Avoiding Unintended Rate Conversion **2-19**
Frame Rebuffering Blocks **2-24**
Buffering with Preservation of the Signal **2-27**
Buffering with Alteration of the Signal **2-30**

Converting Frame Status **2-33**
Frame Status **2-33**
Buffering Sample-Based Signals into Frame-Based Signals **2-33**
Buffering Sample-Based Signals into Frame-Based Signals with Overlap **2-37**
Buffering Frame-Based Signals into Other Frame-Based Signals **2-41**
Buffering Delay and Initial Conditions **2-44**
Unbuffering Frame-Based Signals into Sample-Based Signals **2-45**

Delay and Latency **2-49**
Computational Delay **2-49**
Algorithmic Delay **2-51**
Zero Algorithmic Delay **2-51**
Basic Algorithmic Delay **2-54**
Excess Algorithmic Delay (Tasking Latency) **2-57**
Predicting Tasking Latency **2-59**

3 Filters

Digital Filter Block **3-2**
Overview of the Digital Filter Block **3-2**
Implementing a Lowpass Filter **3-3**
Implementing a Highpass Filter **3-4**
Filtering High-Frequency Noise **3-5**

Specifying Static Filters 3-9
Specifying Time-Varying Filters 3-10
Specifying the SOS Matrix (Biquadratic Filter Coefficients) 3-15

Digital Filter Design Block 3-17
Overview of the Digital Filter Design Block 3-17
Choosing Between Filter Design Blocks 3-18
Creating a Lowpass Filter 3-21
Creating a Highpass Filter 3-23
Filtering High-Frequency Noise 3-25

Filter Realization Wizard 3-31
Overview of the Filter Realization Wizard 3-31
Designing and Implementing a Fixed-Point Filter 3-32
Setting the Filter Structure and Number of Filter Sections 3-48
Optimizing the Filter Structure 3-49

Analog Filter Design Block 3-51

Adaptive Filters 3-53
Creating an Acoustic Environment 3-53
Creating an Adaptive Filter 3-55
Customizing an Adaptive Filter 3-60
Adaptive Filtering Demos 3-64

Multirate Filters 3-66
Filter Banks 3-66
Multirate Filtering Examples 3-74

4 Transforms

Transforming Time-Domain Data into the Frequency Domain 4-2

Transforming Frequency-Domain Data into the Time Domain 4-7

Linear and Bit-Reversed Output Order 4-12
FFT and IFFT Blocks Data Order 4-12
Finding the Bit-Reversed Order of Your Frequency Indices 4-12

Calculating the Channel Latencies Required for Wavelet Reconstruction 4-14
Analyzing Your Model 4-14
Calculating the Group Delay of Your Filters 4-16
Reconstructing the Filter Bank System 4-18
Equalizing the Delay on Each Filter Path 4-18
Updating and Running the Model 4-21
References 4-22

5 Quantizers

Scalar Quantizers 5-2
Analysis and Synthesis of Speech 5-2
Identifying Your Residual Signal and Reflection Coefficients 5-4
Creating a Scalar Quantizer 5-5

Vector Quantizers 5-10
Building Your Vector Quantizer Model 5-10
Configuring and Running Your Model 5-11

6 Statistics, Estimation, and Linear Algebra

Statistics 6-2
Statistics Blocks 6-2

Basic Operations **6-3**
Running Operations **6-4**

Power Spectrum Estimation **6-6**

Linear Algebra **6-7**
Linear Algebra Blocks **6-7**
Linear System Solvers **6-7**
Matrix Factorizations **6-9**
Matrix Inverses **6-11**

7 Working with Fixed-Point Data

Fixed-Point Signal Processing Development **7-2**
Fixed-Point Features **7-2**
Benefits of Fixed-Point Hardware **7-2**
Benefits of Fixed-Point Design with Signal Processing Blockset Software **7-3**
Fixed-Point Signal Processing Applications **7-4**

Concepts and Terminology **7-5**
Fixed-Point Data Types **7-5**
Scaling **7-6**
Precision and Range **7-7**

Arithmetic Operations **7-11**
Modulo Arithmetic **7-11**
Two’s Complement **7-12**
Addition and Subtraction **7-13**
Multiplication **7-14**
Casts **7-17**

Specifying Fixed-Point Attributes **7-22**
Fixed-Point Block Parameters **7-22**
Specifying System-Level Settings **7-25**
Inherit via Internal Rule **7-26**

Example: Selecting and Specifying Data Types for Fixed-Point Blocks **7-37**

Fixed-Point Filtering **7-45**
Fixed-Point Filtering Blocks **7-45**
Filter Implementation Blocks **7-45**
Filter Design and Implementation Blocks **7-46**

8 Getting Started with System Objects

What Are System Objects? **8-2**

Setting Up and Running System Objects **8-3**
Creating an Instance of a System Object **8-3**
Using Methods to Run System Objects **8-6**
Finding Help and Demos for System Objects **8-8**

Using System Objects with the Embedded MATLAB Subset **8-9**
Considerations for Using System Objects with the Embedded MATLAB Subset **8-9**
Using System Objects with Embedded MATLAB Coder . . **8-11**
Using System Objects with the Embedded MATLAB Function Block **8-12**
Using System Objects with Embedded MATLAB MEX . . . **8-12**

9 Using Signal Processing System Objects

What Are Signal Processing System Objects? **9-2**

Generating Code for Signal Processing System Objects **9-3**

Working with Signals and Fixed-Point Data **9-5**
What Are Sample- and Frame-Based Processing? **9-5**
Working with Fixed-Point Data **9-10**

Example: Using System Objects in Signal Processing Applications: Filtering an Audio Stream **9-15**

Index

Working with Signals

This chapter helps you understand how sample-based and frame-based signals are represented in the Simulink® environment. You learn how to create single-channel and multichannel sample-based and frame-based signals. You also learn how to extract single-channel signals from multichannel signals. Lastly you explore how to import signals into signal processing models and export signals to the MATLAB® workspace.

- "Discrete-Time Signals" on page 1-2
- "Continuous-Time Signals" on page 1-11
- "Sample-Based Signals" on page 1-13
- "Frame-Based Signals" on page 1-15
- "Creating Sample-Based Signals" on page 1-19
- "Creating Frame-Based Signals" on page 1-25
- "Creating Multichannel Sample-Based Signals" on page 1-32
- "Creating Multichannel Frame-Based Signals" on page 1-38
- "Deconstructing Multichannel Sample-Based Signals" on page 1-42
- "Deconstructing Multichannel Frame-Based Signals" on page 1-49
- "Importing and Exporting Sample-Based Signals" on page 1-58
- "Importing and Exporting Frame-Based Signals" on page 1-70
- "Displaying Time-Domain Data" on page 1-79
- "Displaying Frequency-Domain Data" on page 1-100

Discrete-Time Signals

In this section...

"Time and Frequency Terminology" on page 1-2

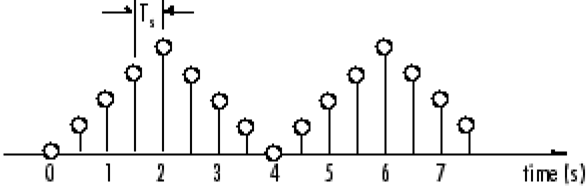
"Recommended Settings for Discrete-Time Simulations" on page 1-4

"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 Signal Processing Blockset™ 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`

Configuration Parameters: untitled/Configuration

Select:
Solver
Data Import/Export
Optimization
Diagnostics
Sample Time
Data Integrity
Conversion
Connectivity
Compatibility
Model Referencing
Hardware Implementation
Model Referencing
Real-Time Workshop
Comments
Symbols
Custom Code
Debug
Interface

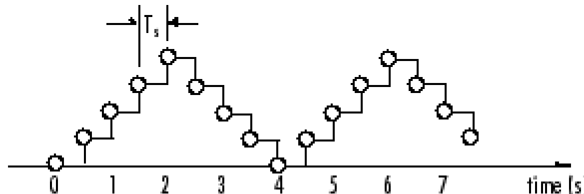
Simulation time
Start time: 0.0 Stop time: 10.0

Solver options
Type: Fixed-step Solver: discrete (no continuous states)
Periodic sample time constraint: Unconstrained
Fixed step size (fundamental sample time): auto
Tasking mode for periodic sample times: SingleTasking
Higher priority value indicates higher task priority

OK Cancel Help Apply

You can automatically set the above solver options for all new models by running the `dspstartup.m` file. See “Configuring the Simulink Environment for Signal Processing Models” in the *Signal Processing Blockset 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 2-57 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.

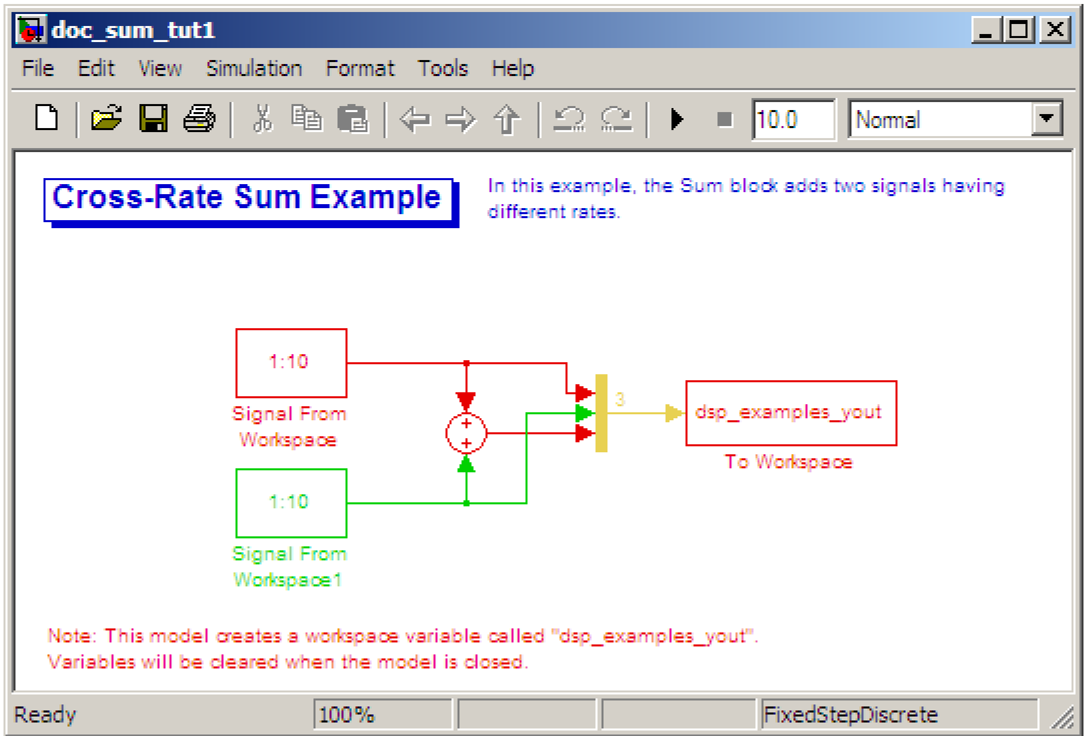
Cross-Rate Operations

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 "Converting Sample and Frame Rates" on page 2-11 for more information. 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 `doc_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...

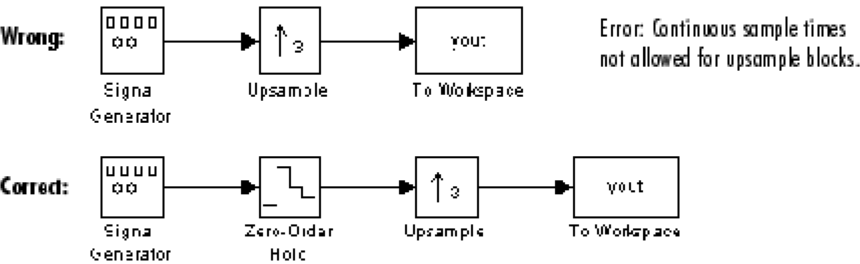
"Continuous-Time Source Blocks" on page 1-11

"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 Signal Processing 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 Signal Processing Blockset software accept continuous-time signals, and all nonsource blocks inherit the sample period of the input. Therefore, continuous-time inputs generate continuous-time

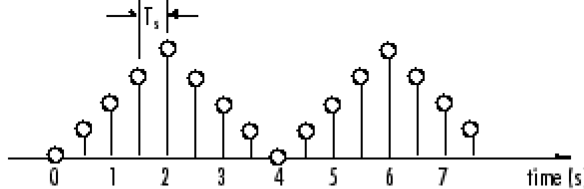
outputs. Blocks that are not capable of accepting continuous-time signals include the Digital Filter, FIR Decimation, FIR Interpolation blocks.

Sample-Based Signals

In this section...
“Sample-Based Single Channel Signals” on page 1-13
“Sample-Based Multichannel Signals” on page 1-13

Sample-Based Single Channel Signals

Signals can be sample-based or frame-based, single channel or multichannel. The following figure shows a discrete-time signal. If this signal is propagated through a model sample-by-sample, rather than in batches of samples, it is called a sample-based signal. It is also single-channel signal, because there is only one independent sequence of numbers.

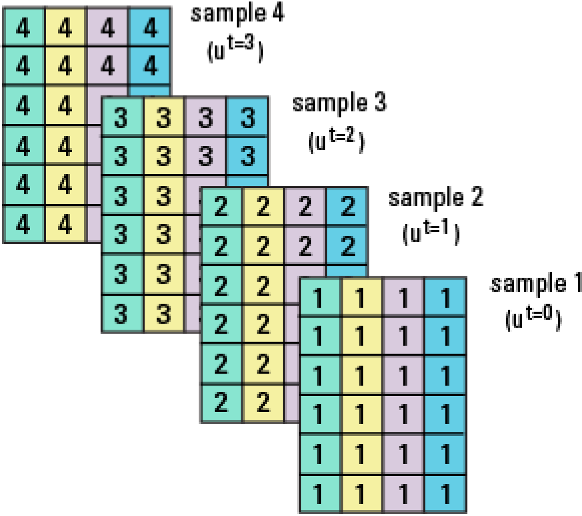


The representation of single-channel signals is actually a special case of the general multichannel signal.

Sample-Based Multichannel Signals

Sample-based multichannel signals are represented as matrices. An M-by-N sample-based matrix represents M*N independent channels, each containing a single value. In other words, each matrix element represents one sample from a distinct channel.

As an example, consider the 24-channel (6-by-4) sample-based signal in the figure below, where $u^{t=0}$ is the first matrix in the series, $u^{t=1}$ is the second, $u^{t=2}$ is the third, and so on.



The signal in channel 1 is composed of the following sequence:

$$u_{11}^{t=0}, u_{11}^{t=1}, u_{11}^{t=2}, \ldots$$

Similarly, channel 9 (counting down the columns) contains the following sequence:

$$u_{32}^{t=0}, u_{32}^{t=1}, u_{32}^{t=2}, \ldots$$

In practice, signal samples are frequently transmitted in batches, or frames, and several channels of data are often transmitted simultaneously in order to accelerate simulations. Hence, most signals are frame-based and multichannel signals.

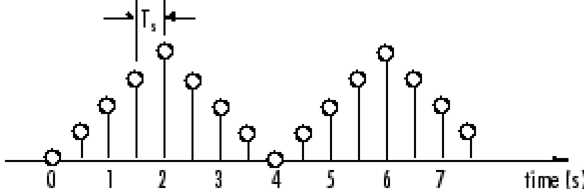
Frame-Based Signals

In this section...

"Frame-Based Single Channel Signals" on page 1-15
"Frame-Based Multichannel Signals" on page 1-15
"Benefits of Frame-Based Processing" on page 1-16

Frame-Based Single Channel Signals

Signals can be sample-based or frame-based, single channel or multichannel. The following figure shows a discrete-time signal. If this signal is propagated through a model in batches of samples, it is called a frame-based signal. It is also a single-channel signal, because there is only one independent sequence of numbers.

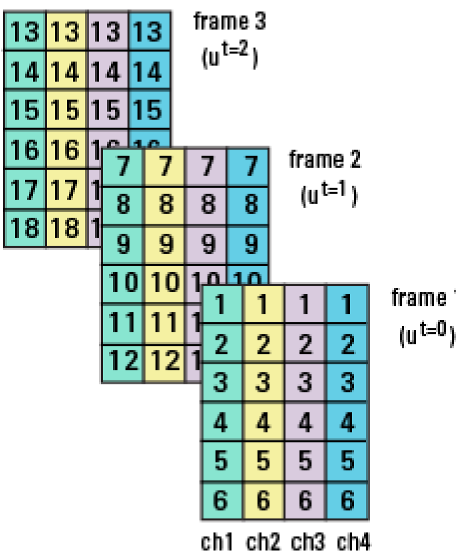


Frame-based single channel signals are represented as vectors. An M-by-1 frame-based vector represents M consecutive samples from a single channel. In other words, each matrix row represents one sample, or time slice, from one distinct channel.

Frame-Based Multichannel Signals

Frame-based multichannel signals are represented as matrices. An M-by-N frame-based matrix represents M consecutive samples from each of N independent channels. In other words, each matrix row represents one sample, or time slice, from N distinct signal channels, and each matrix column represents M consecutive samples from a single channel.

Consider a sequence of frame matrices, where $u^{t=0}$ is the first matrix in a series, $u^{t=1}$ is the second, $u^{t=2}$ is the third, and so on.



The signal in channel 1 is the following sequence:

$$u_{11}^{t=0}, u_{21}^{t=0}, u_{31}^{t=0}, \ldots, u_{M1}^{t=0}, u_{11}^{t=1}, u_{21}^{t=1}, u_{31}^{t=1}, \ldots, u_{M1}^{t=1}, u_{11}^{t=2}, u_{21}^{t=2}, \ldots$$

Similarly, the signal in channel 3 is the following sequence:

$$u_{13}^{t=0}, u_{23}^{t=0}, u_{33}^{t=0}, \ldots, u_{M3}^{t=0}, u_{13}^{t=1}, u_{23}^{t=1}, u_{33}^{t=1} \ldots, u_{M3}^{t=1}, u_{13}^{t=2}, u_{23}^{t=2}, \ldots$$

Benefits of Frame-Based Processing

Frame-based processing is an established method of accelerating both real-time systems and simulations.

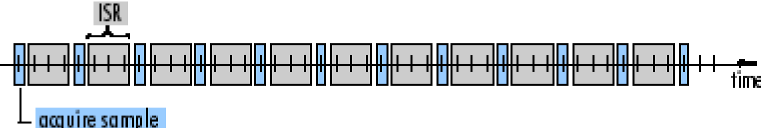
Accelerating Real-Time Systems

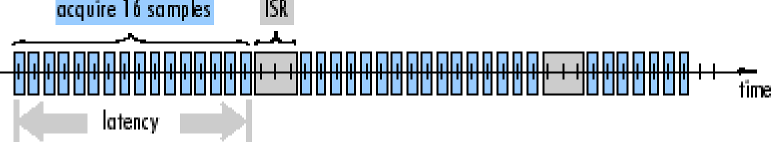
Frame-based data is a common format in real-time systems. Data acquisition hardware often operates by accumulating a large number of signal samples

at a high rate, and propagating these samples to the real-time system as a block of data. This maximizes the efficiency of the system by distributing the fixed process overhead across many samples; the “fast” data acquisition is suspended by “slow” interrupt processes after each frame is acquired, rather than after each individual sample.

The figure below illustrates how throughput is increased by frame-based data acquisition. 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. The frame-based throughput rate is therefore many times higher than the sample-based alternative.





It’s important to note 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, it is possible to select frame sizes that improve throughput without creating unacceptable latencies. For more information, see “Delay and Latency” on page 2-49.

Accelerating Simulations

The simulation of your model also benefits from frame-based processing. In this case, it is the overhead of block-to-block communications that is reduced by propagating frames rather than individual samples.

Creating Sample-Based Signals

In this section...
"Using the Constant Block" on page 1-19
"Using the Signal from Workspace Block" on page 1-21

Using the Constant Block

A constant sample-based signal has identical successive samples. The Signal Processing 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 Signal Processing Sources library, click-and-drag a Constant block into the model.

3. From the Signal Processing 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.

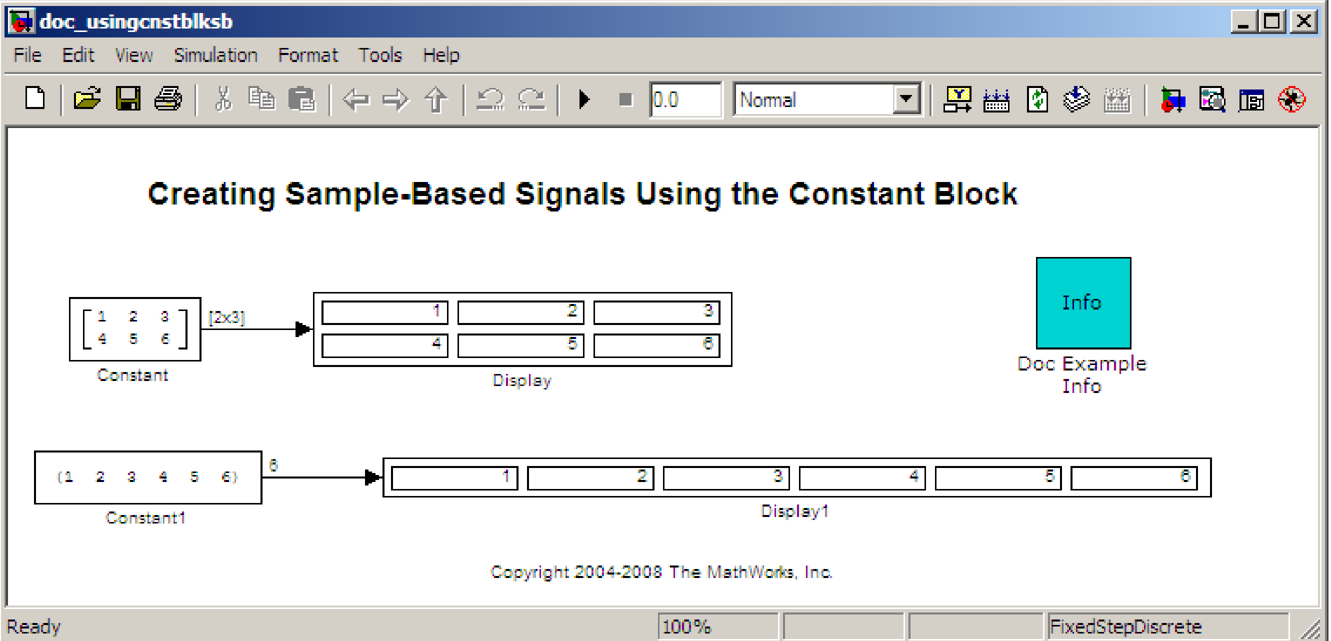
Creating a 1-D Vector Signal

You can create a 1-D vector signal 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 `doc_usingcnstblksb` at the MATLAB command line.



The Constant1 block generates a length-6 1-D vector signal. This means that the output is not a matrix. However, most nonsource signal processing blocks interpret a length-M 1-D vector as an M-by-1 matrix (column vector).

Note A 1-D vector signal must always be sample based.

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

3. From the Signal Processing 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:

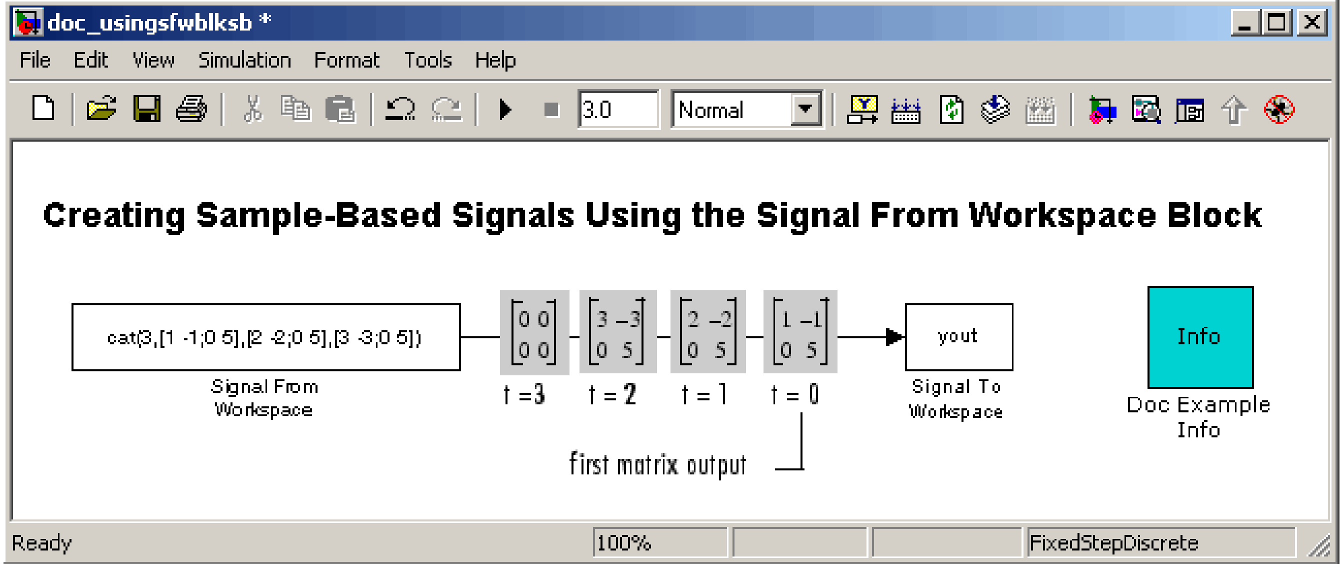
   - Channel 1: 1, 2, 3, 0, 0,...
   - Channel 2: -1, -2, -3, 0, 0,...
   - Channel 3: 0, 0, 0, 0, 0,...
   - Channel 4: 5, 5, 5, 0, 0,...

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 `doc_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.

Creating Frame-Based Signals

In this section...

"Using the Sine Wave Block" on page 1-25

"Using the Signal from Workspace Block" on page 1-28

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. The Signal Processing Sources library provides the following blocks for automatically generating common frame-based signals:

- Chirp
- Discrete Impulse
- Constant
- Multiphase Clock
- N-Sample Enable
- Signal From Workspace
- Sine Wave

For information about the specific functionality of these blocks, see their respective block reference pages.

One of the most commonly used blocks in the Signal Processing 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 Signal Processing 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 Signal Processing 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**.

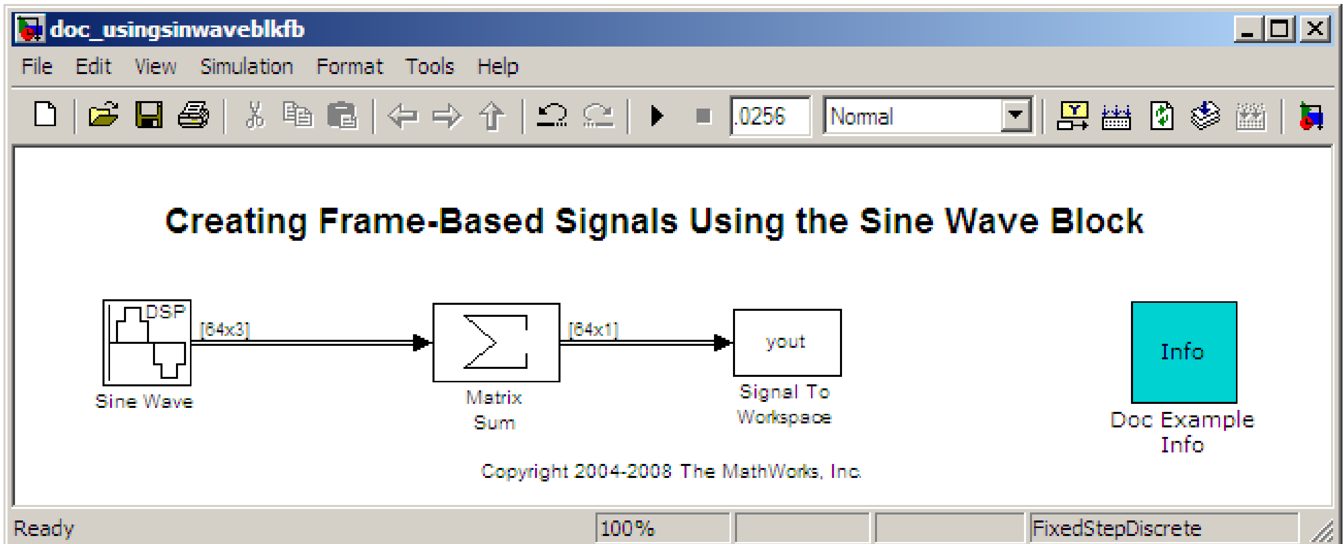
   You have now successfully created a three-channel frame-based signal using the Sine Wave block. The rest of this procedure describes how to add these three sinusoids together.

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.

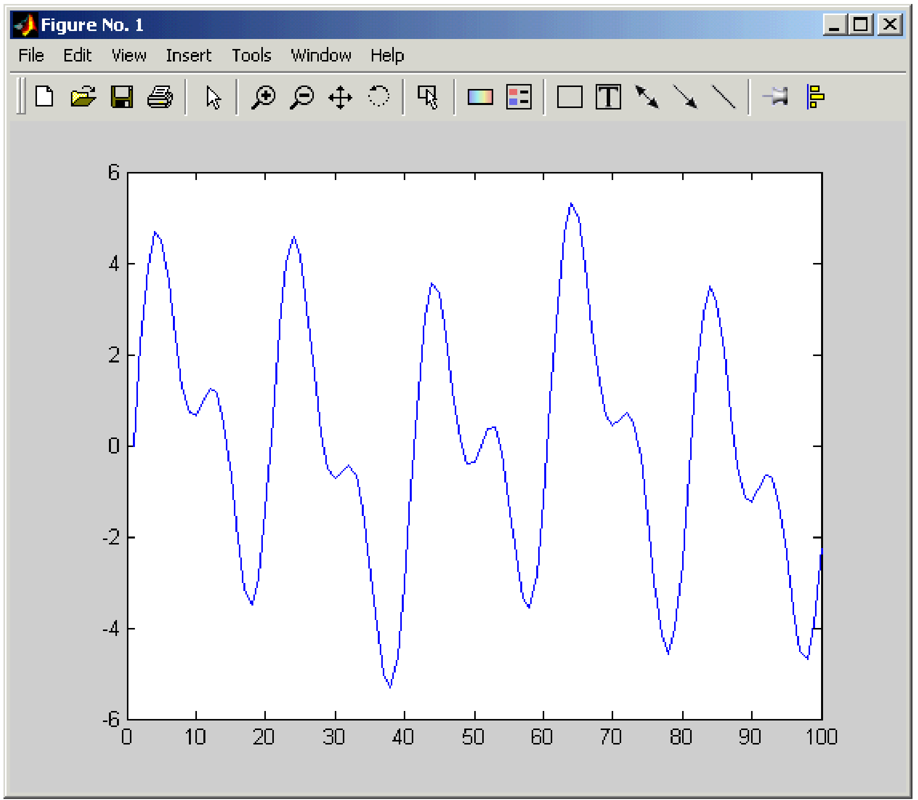
Your model should now look similar to the following figure. You can also open the model by typing `doc_usingsinwaveblkfb` at the MATLAB command line.



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.

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

3. From the Signal Processing 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:

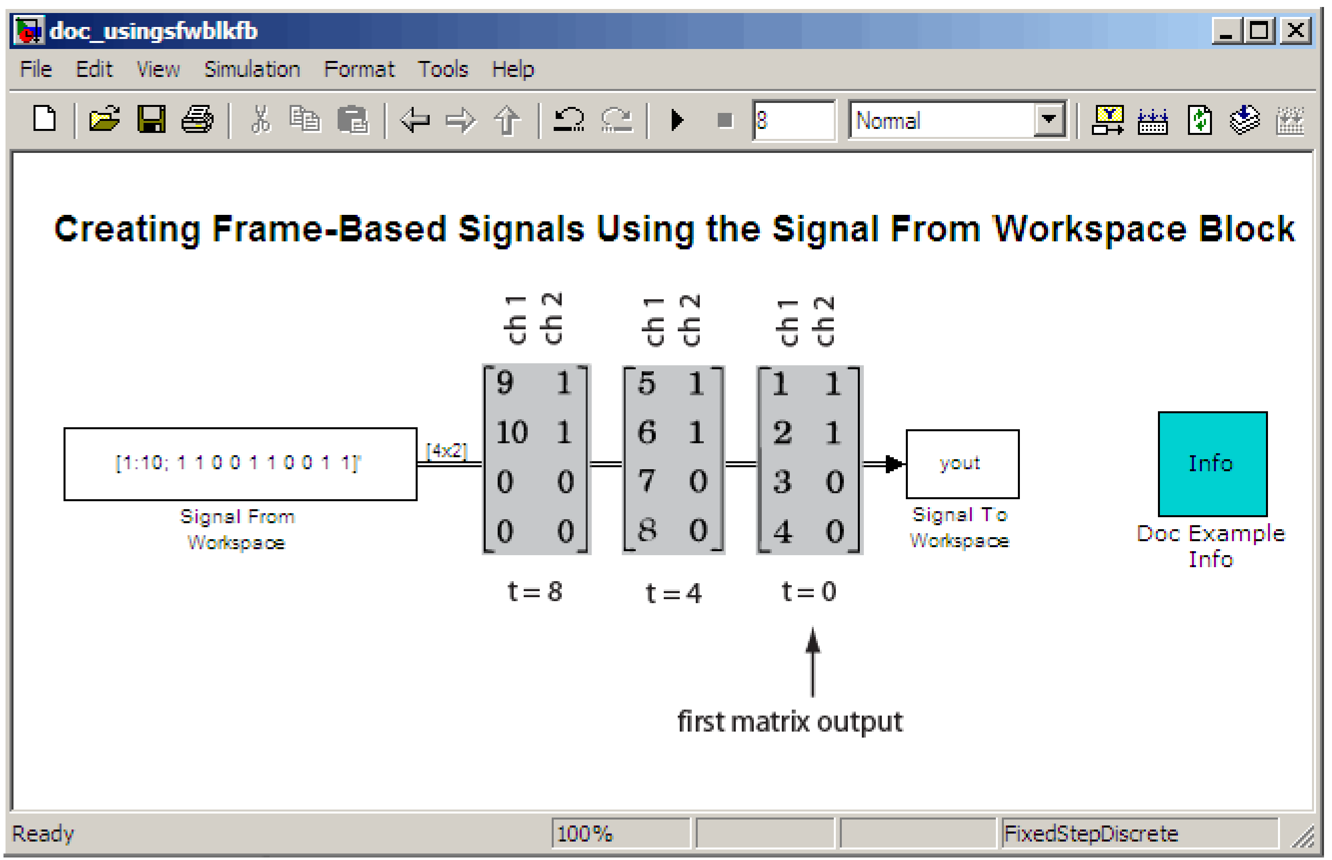
   - Channel 1: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 0, 0,...
   - Channel 2: 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0,...

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 `doc_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.

Creating Multichannel Sample-Based Signals

In this section...

"Multichannel Sample-Based Signals" on page 1-32

"Combining Single-Channel Sample-Based Signals" on page 1-32

"Combining Multichannel Sample-Based Signals" on page 1-35

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*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.

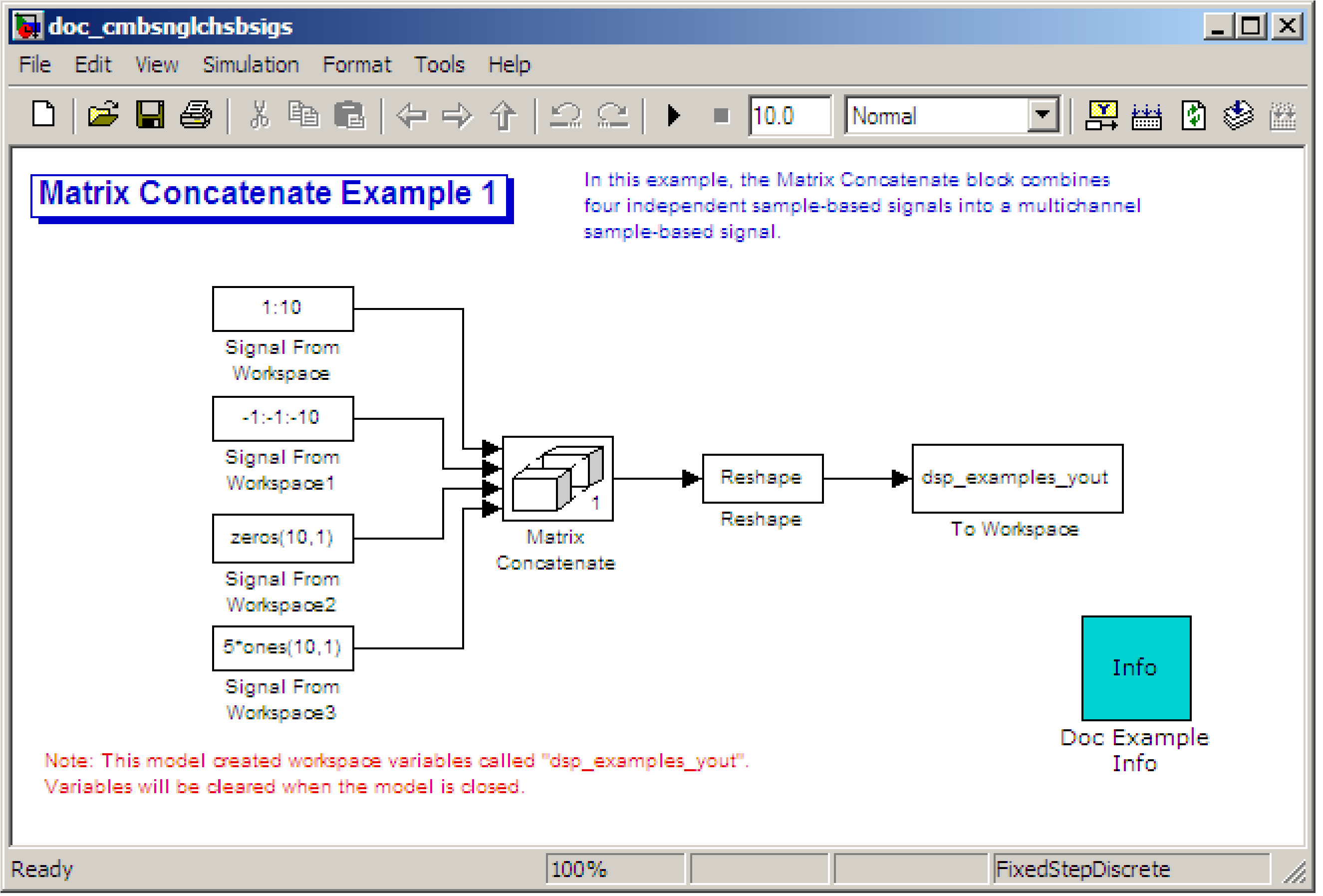
Combining Single-Channel Sample-Based Signals

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

```
doc_cmbsnglchsbsigs
```

at the MATLAB command line.



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

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

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

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

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

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

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

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

8 Run the model.

Four independent 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.

$$\begin{bmatrix}1\\2\\3\\4\end{bmatrix} \Rightarrow \text{Reshape} \Rightarrow \begin{bmatrix}1 & 3\\2 & 4\end{bmatrix}$$

Reshape

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`.

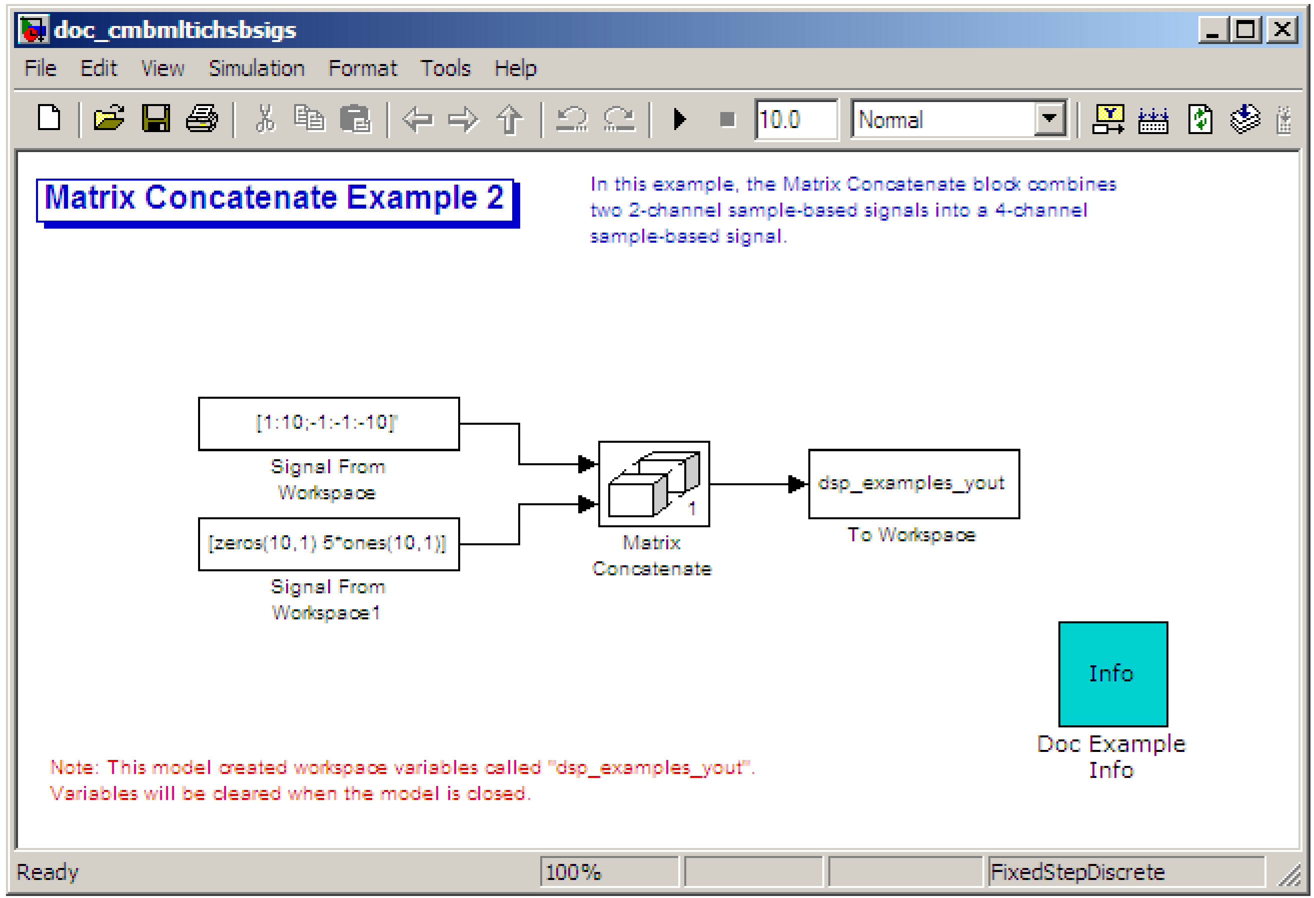
Combining Multichannel Sample-Based Signals

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

   ```
   doc_cmbmltichsbsigs
   ```

   at the MATLAB command line.



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

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

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

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

- **Concatenate dimension** = 1

5 Run the model.

   The model combines both two-channel 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.

Creating Multichannel Frame-Based Signals

In this section...

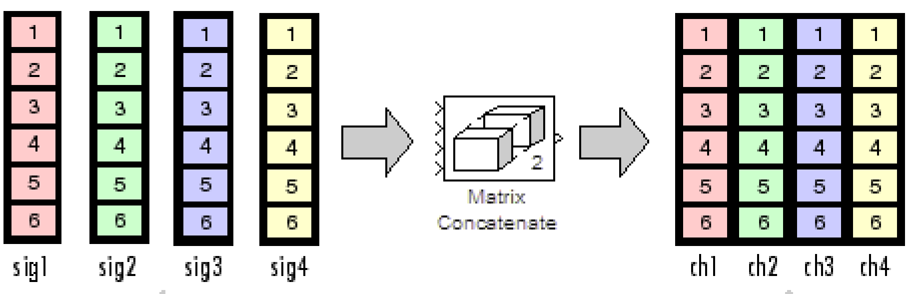
"Multichannel Frame-Based Signals" on page 1-38

"Combining Frame-Based Signals" on page 1-39

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.



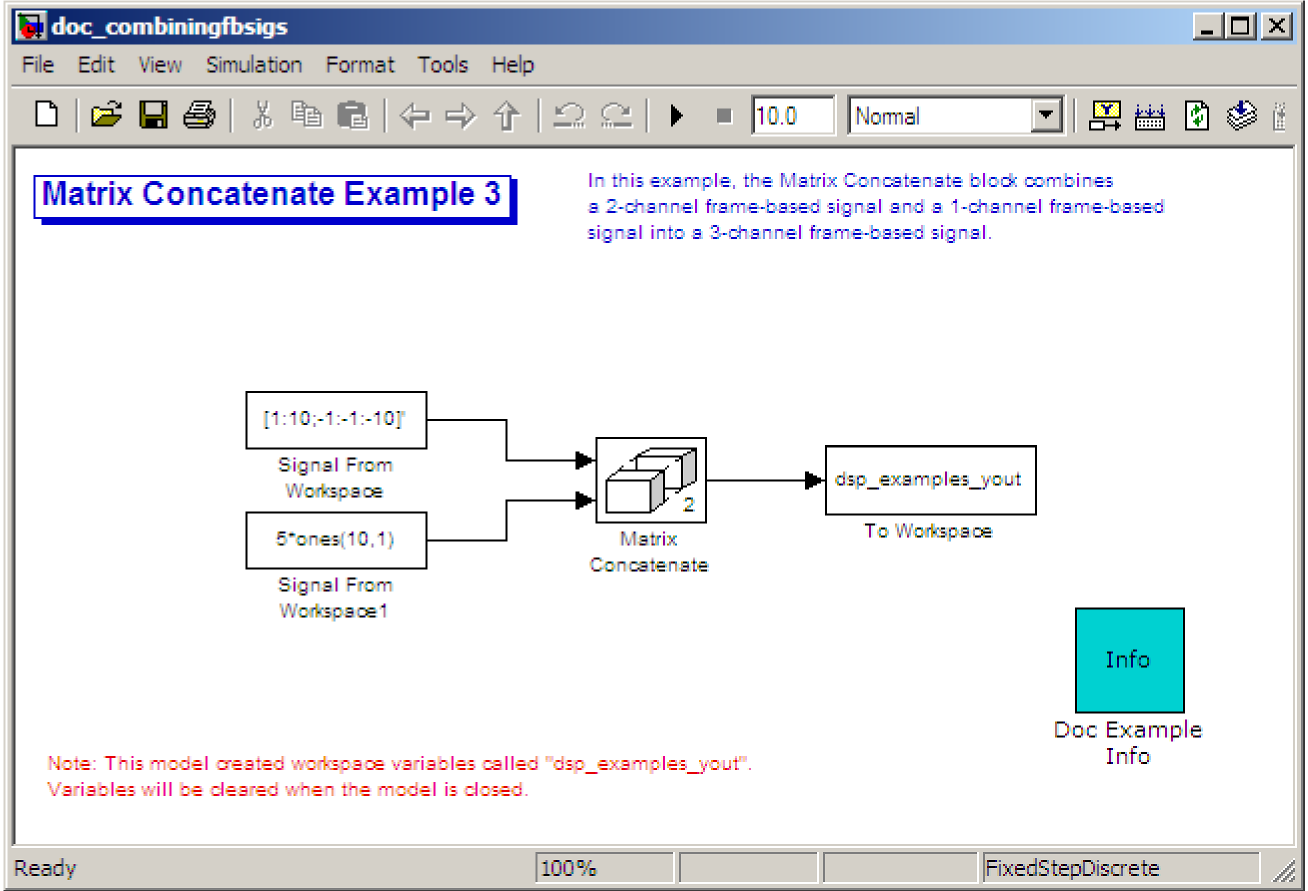
Combining Frame-Based Signals

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

   ```
   doc_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.

Deconstructing Multichannel Sample-Based Signals

In this section...

"Splitting Multichannel Sample-Based Signals into Individual Signals" on page 1-42

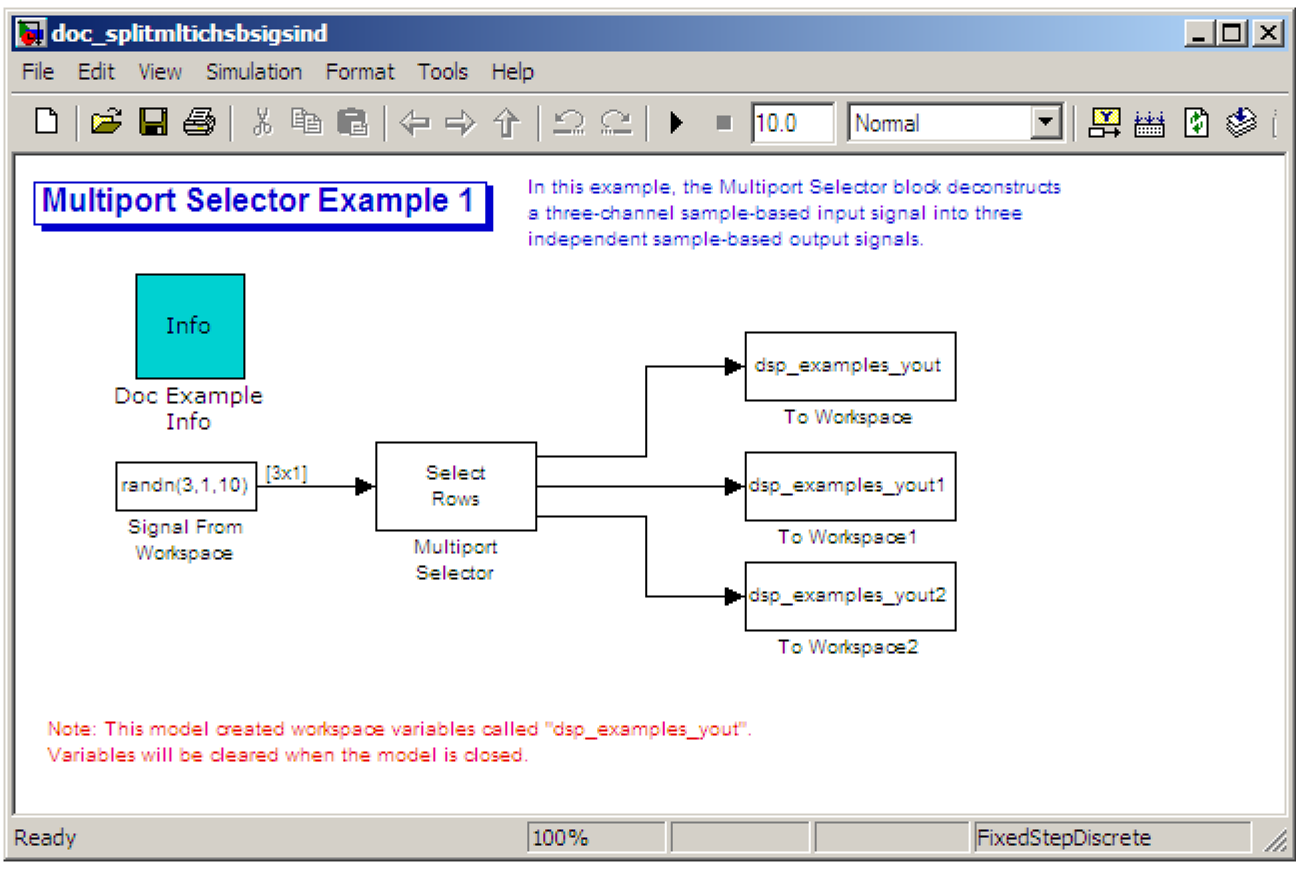
"Splitting Multichannel Sample-Based Signals into Several Multichannel Signals" on page 1-45

Splitting 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 `doc_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.

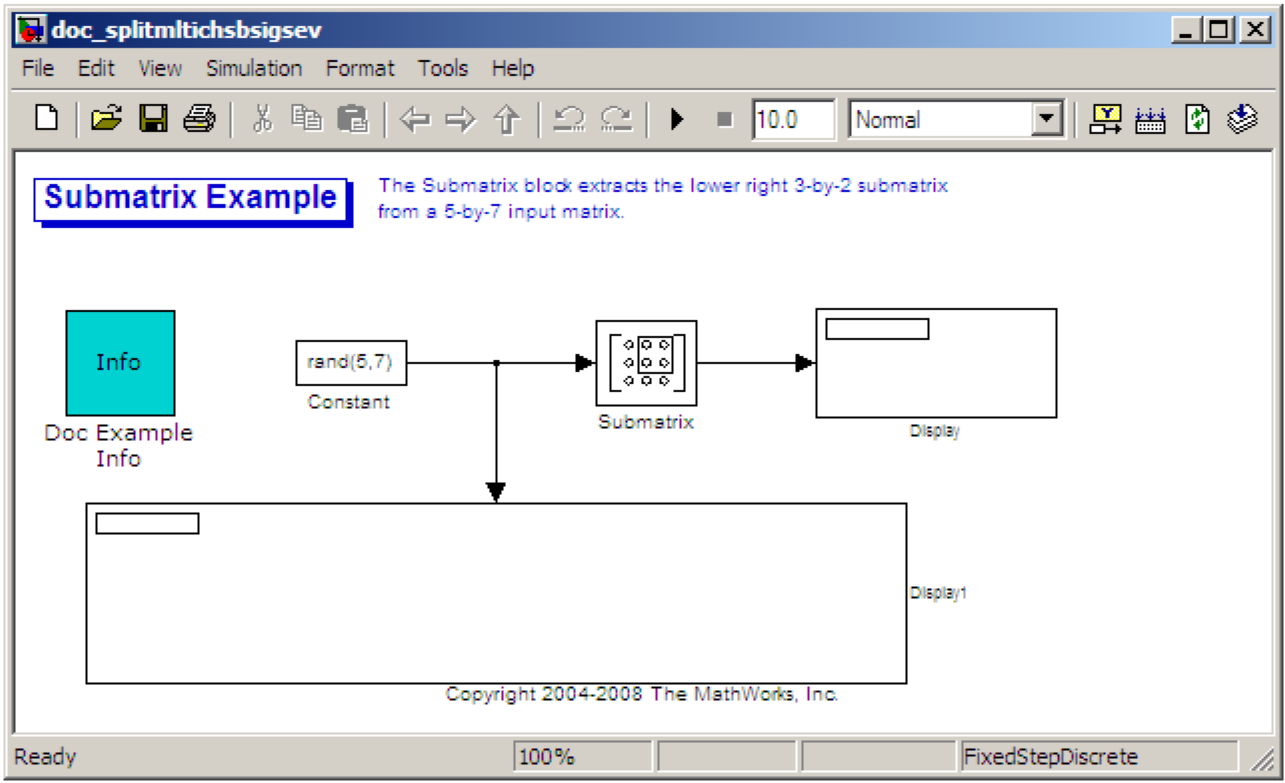
You have now successfully created three, single-channel sample-based signals from a multichannel sample-based signal using a Multiport Selector block.

Splitting 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, Multiport 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 `doc_splitmltichsbsigsev` at the MATLAB command line.



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

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

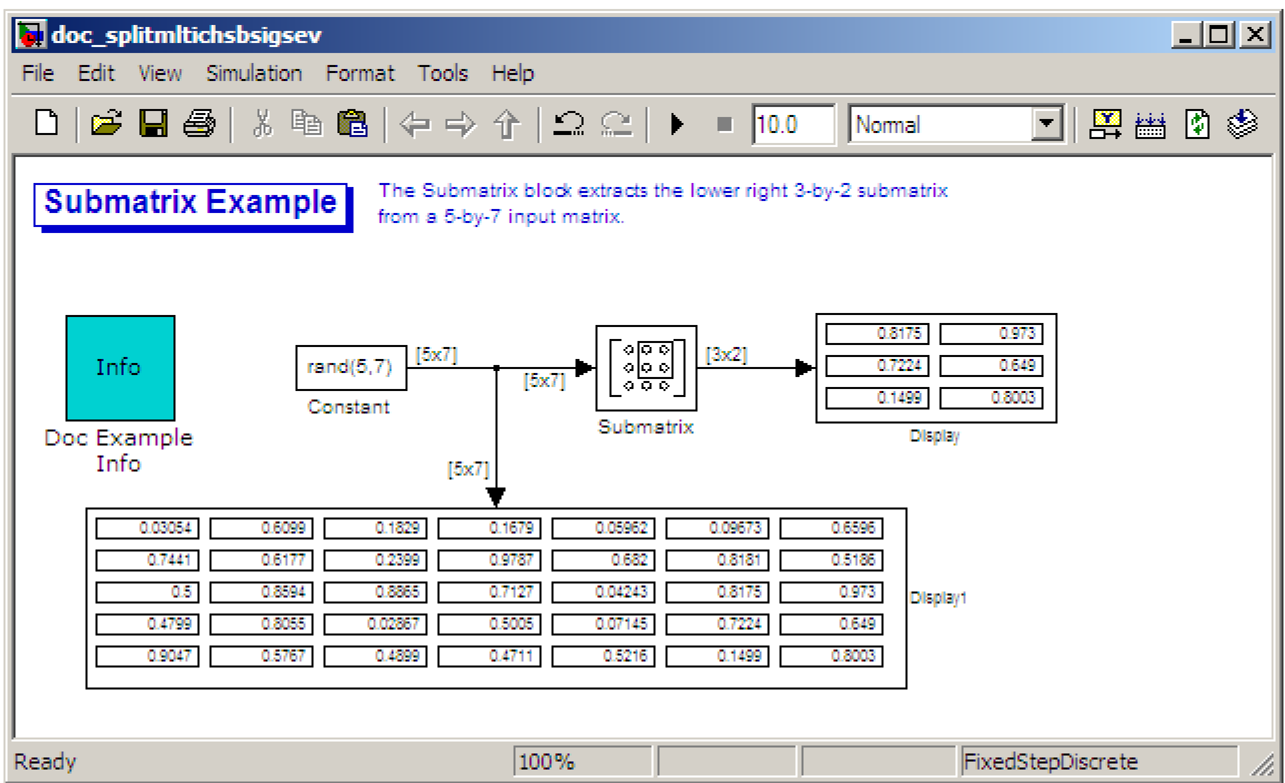
4 Double-click the Submatrix block. Set the block parameters as follows, and then click **OK**:

- **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.

Deconstructing Multichannel Frame-Based Signals

In this section...

"Splitting Multichannel Frame-Based Signals into Individual Signals" on page 1-49

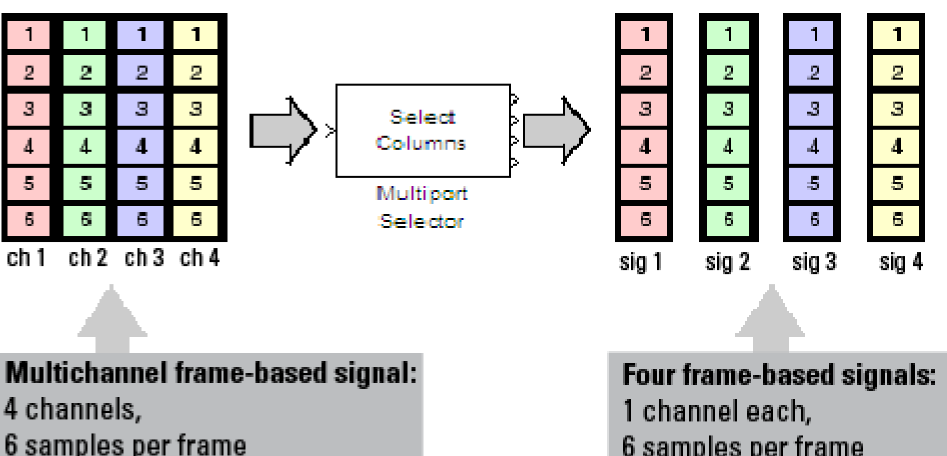
"Reordering Channels in Multichannel Frame-Based Signals" on page 1-54

Splitting 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.

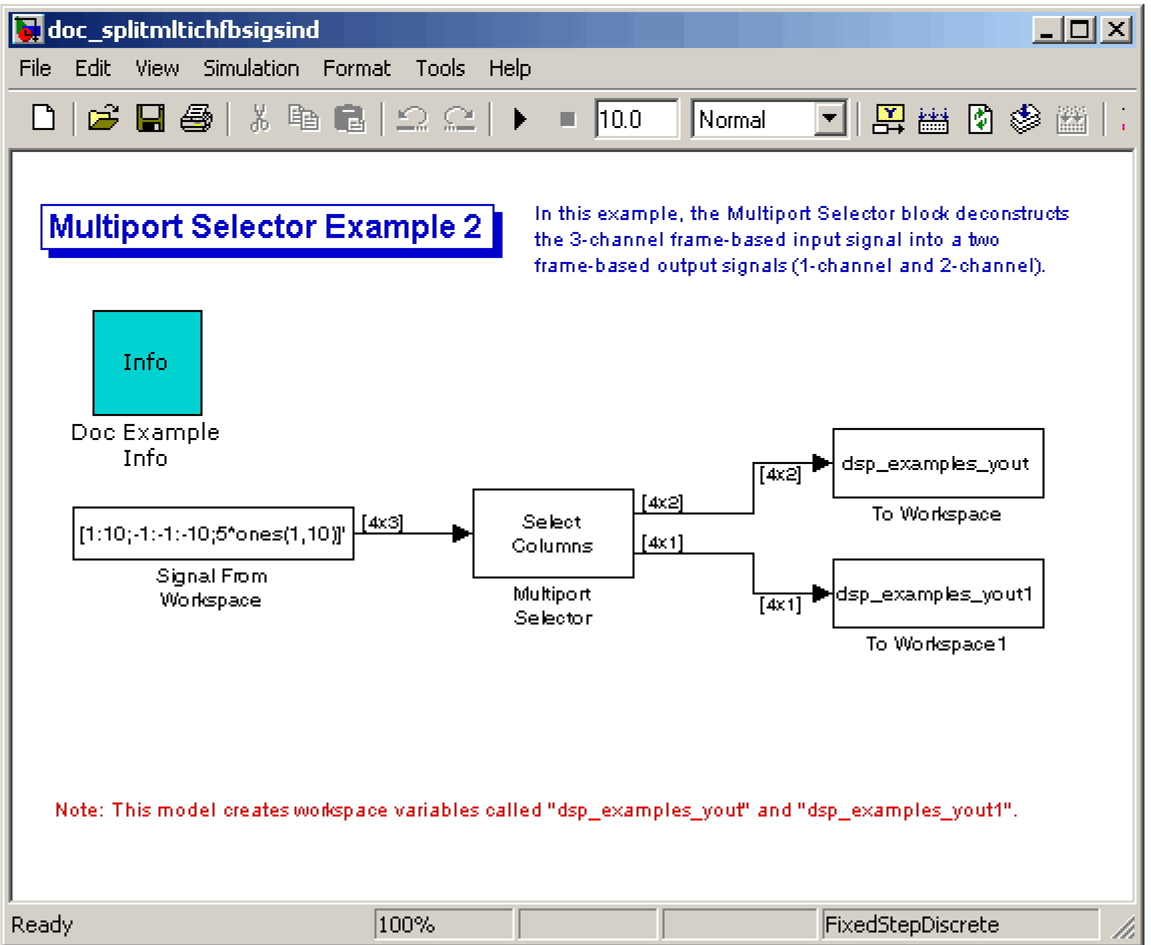
The figure below is a graphical representation of this process.



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
`doc_splitmltichfbsigsind`

   at the MATLAB command line.



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

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

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

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

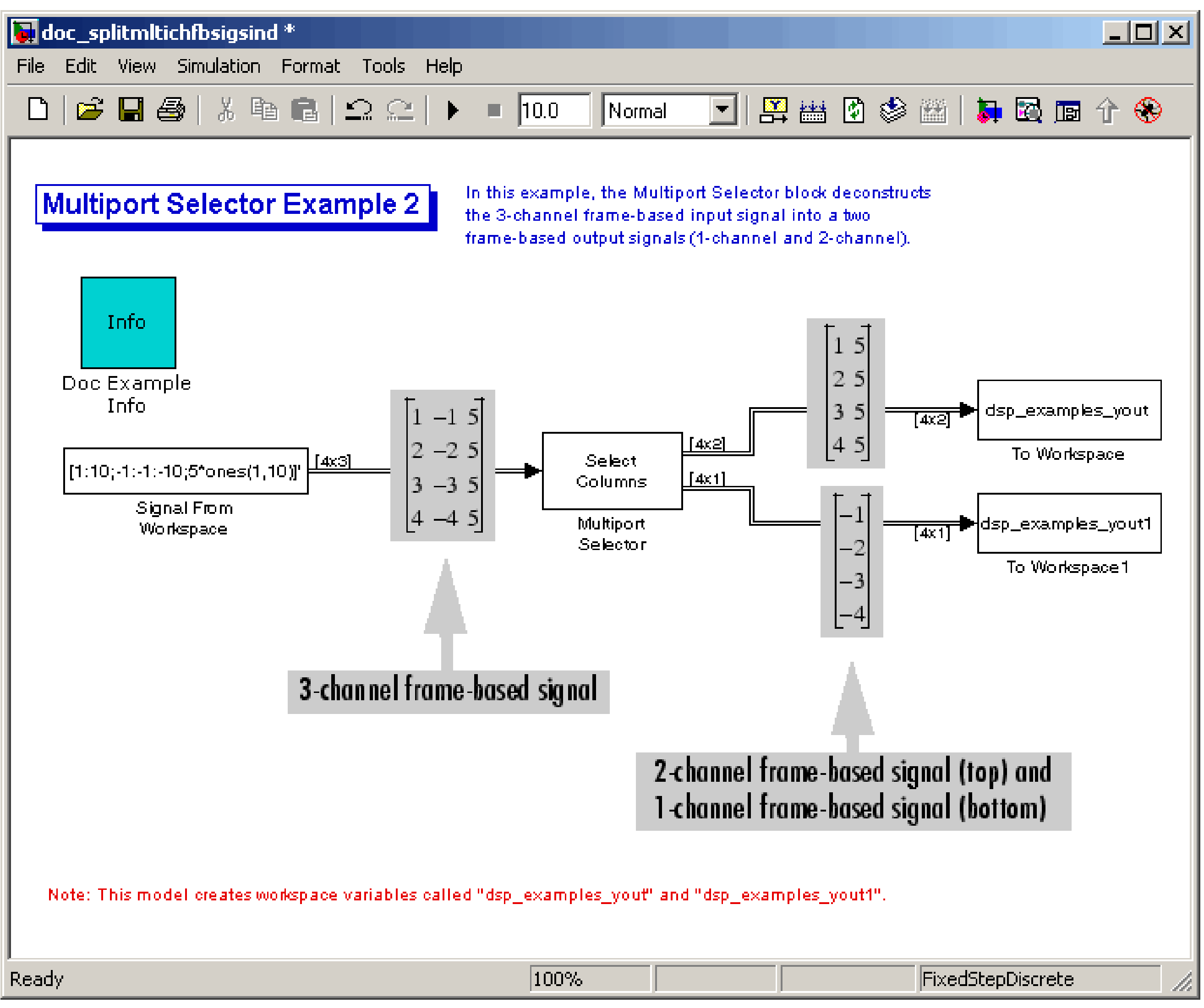
4 Double-click the Multiport Selector block. Set the block parameters as follows, and then click **OK**:

- **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.

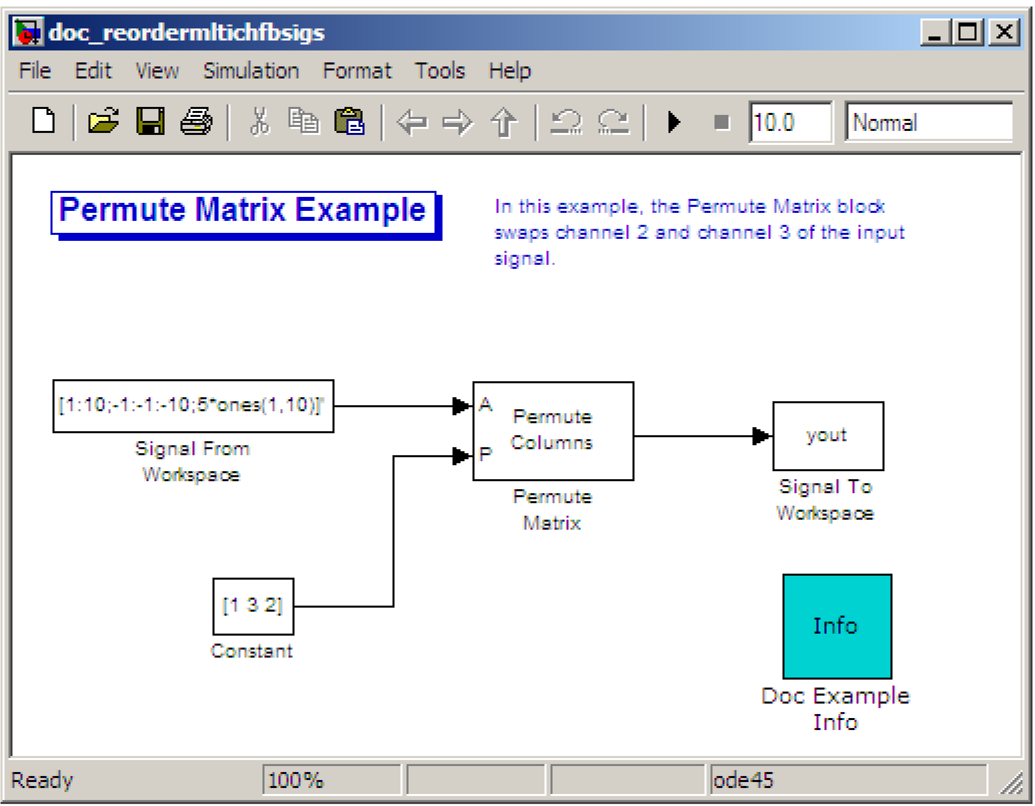
You have now successfully created a single-channel and a two-channel frame-based signal from a multichannel frame-based signal using the Multiport Selector block.

Reordering 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 Signal Processing Blockset blocks have the ability to process the interaction of channels. Typically, Signal Processing Blockset 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 `doc_reordermltichfbsigs` at the MATLAB command line.



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

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

Based on these parameters, the Signal From Workspace block outputs a three-channel, 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.

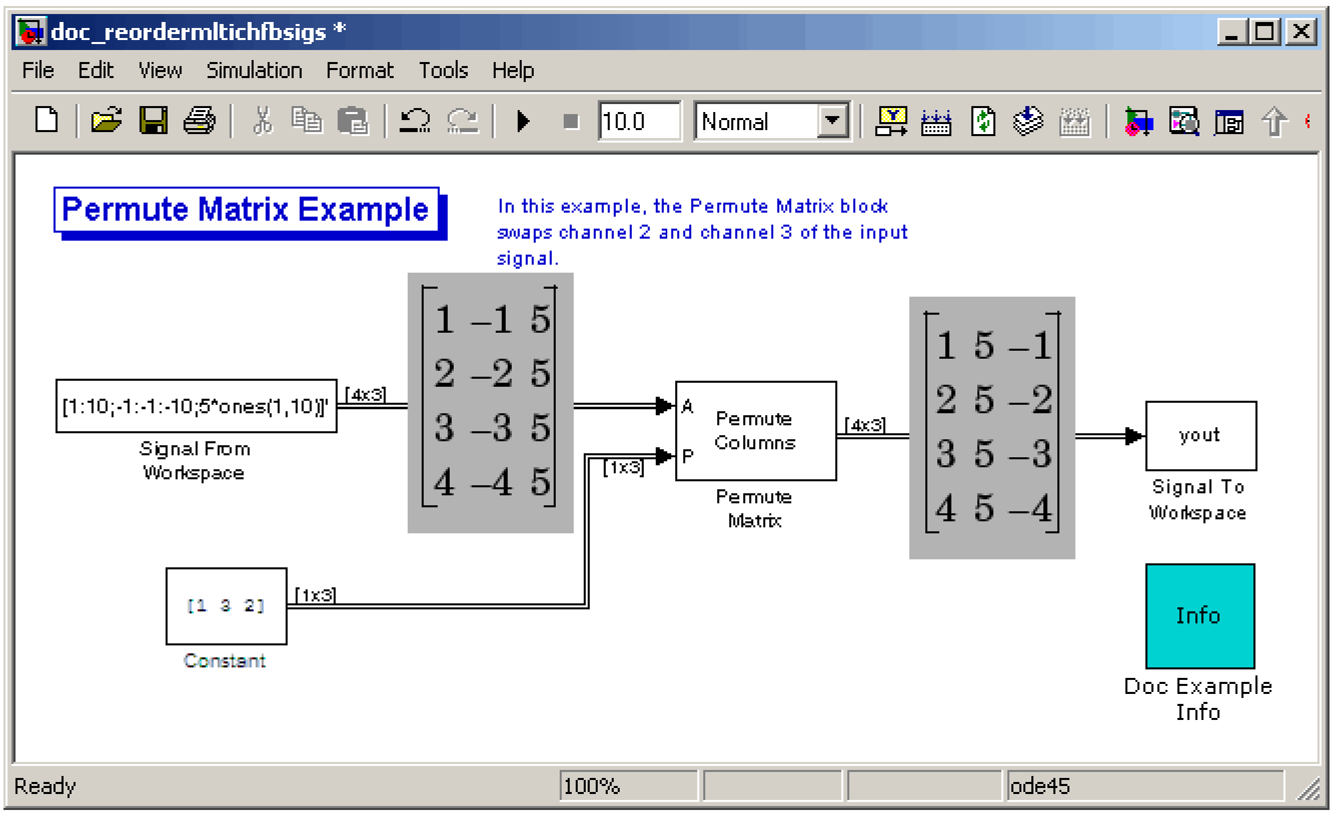
5 Double-click the Permute Matrix block. Set the block parameters as follows, and then click **OK**:

- **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.

Importing and Exporting Sample-Based Signals

In this section...

"Importing Sample-Based Vector Signals" on page 1-58

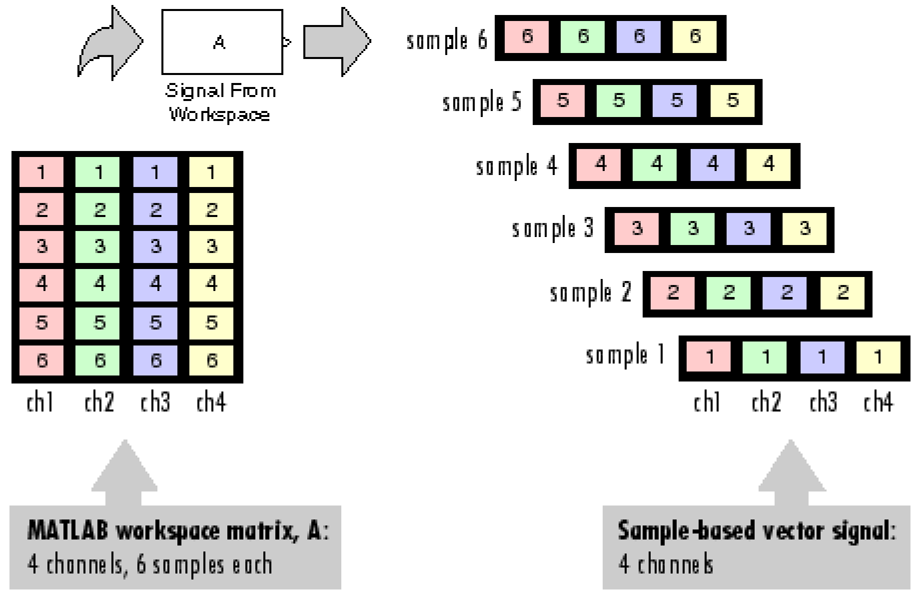
"Importing Sample-Based Matrix Signals" on page 1-61

"Exporting Sample-Based Signals" on page 1-65

Importing Sample-Based Vector Signals

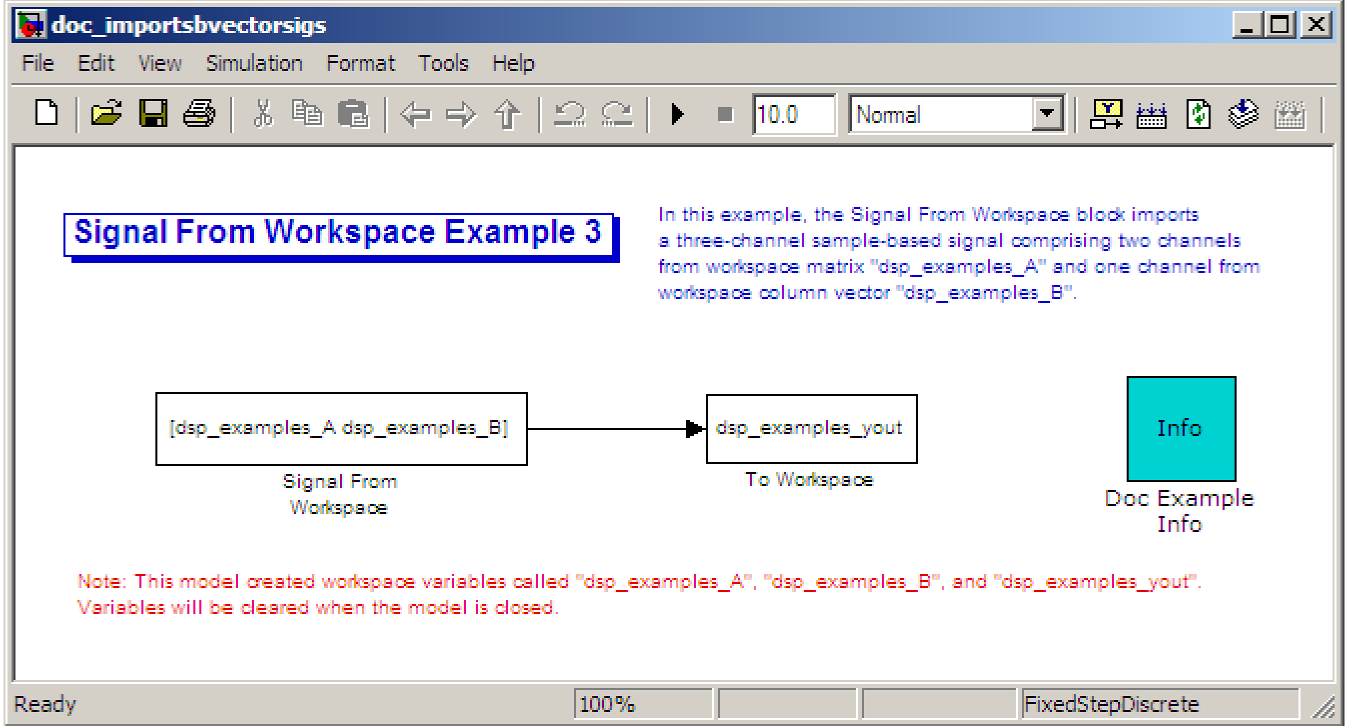
The Signal From Workspace block generates a sample-based vector signal when the variable or expression in the **Signal** parameter is a matrix and the **Samples per frame** parameter is set to 1. Each column of the input matrix represents a different channel. Beginning with the first row of the matrix,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

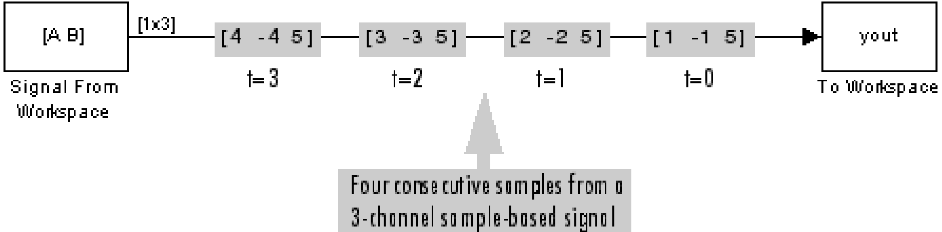
- **Signal** = `[A B]`
- **Sample time** = `1`
- **Samples per frame** = `1`
- **Form output after final data value** = `Setting to zero`

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.



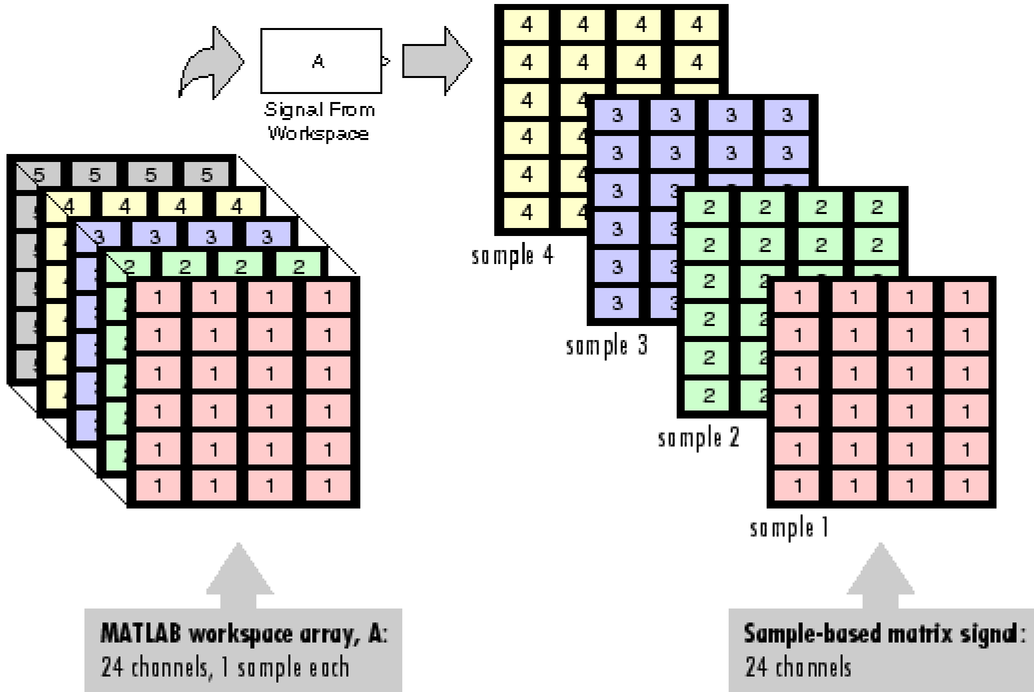
The first row of the input matrix `[A B]` is output at time `t=0`, the second row of the input matrix is output at time `t=1`, and so on.

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

Importing Sample-Based Matrix Signals

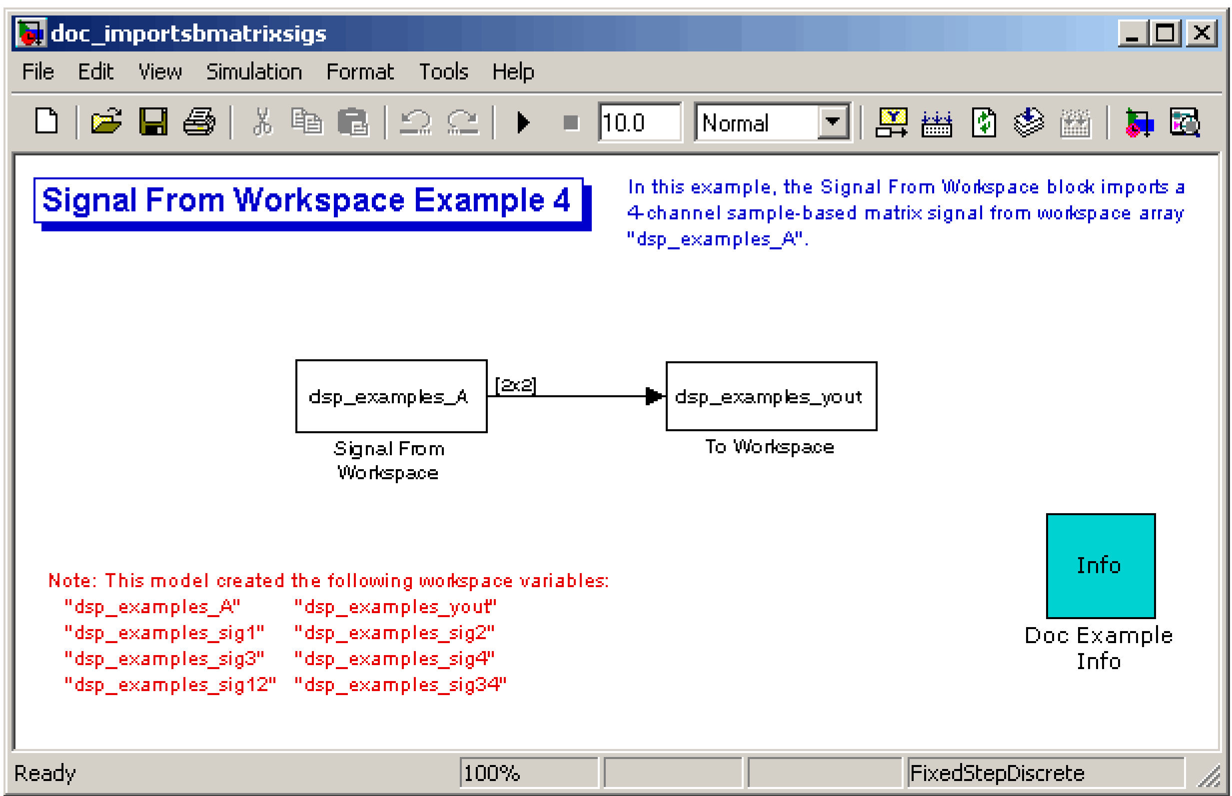
The Signal From Workspace block generates a sample-based matrix signal when the variable or expression in the **Signal** parameter is a three-dimensional array and the **Samples per frame** parameter is set to `1`. Beginning with the first page of the array, the block outputs a single page of the array 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 `doc_importsbmatrixsigs` at the MATLAB command line.



Also, the following variables are loaded into the MATLAB workspace:

```
Fs                  1x1         8        double array
dsp_examples_A      2x2x100     3200     double array
dsp_examples_sig1   1x1x100     800      double array
dsp_examples_sig12  1x2x100     1600     double array
dsp_examples_sig2   1x1x100     800      double array
dsp_examples_sig3   1x1x100     800      double array
dsp_examples_sig34  1x2x100     1600     double array
```

```
dsp_examples_sig4       1x1x100     800       double array
mtlb                    4001x1      32008     double array
```

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

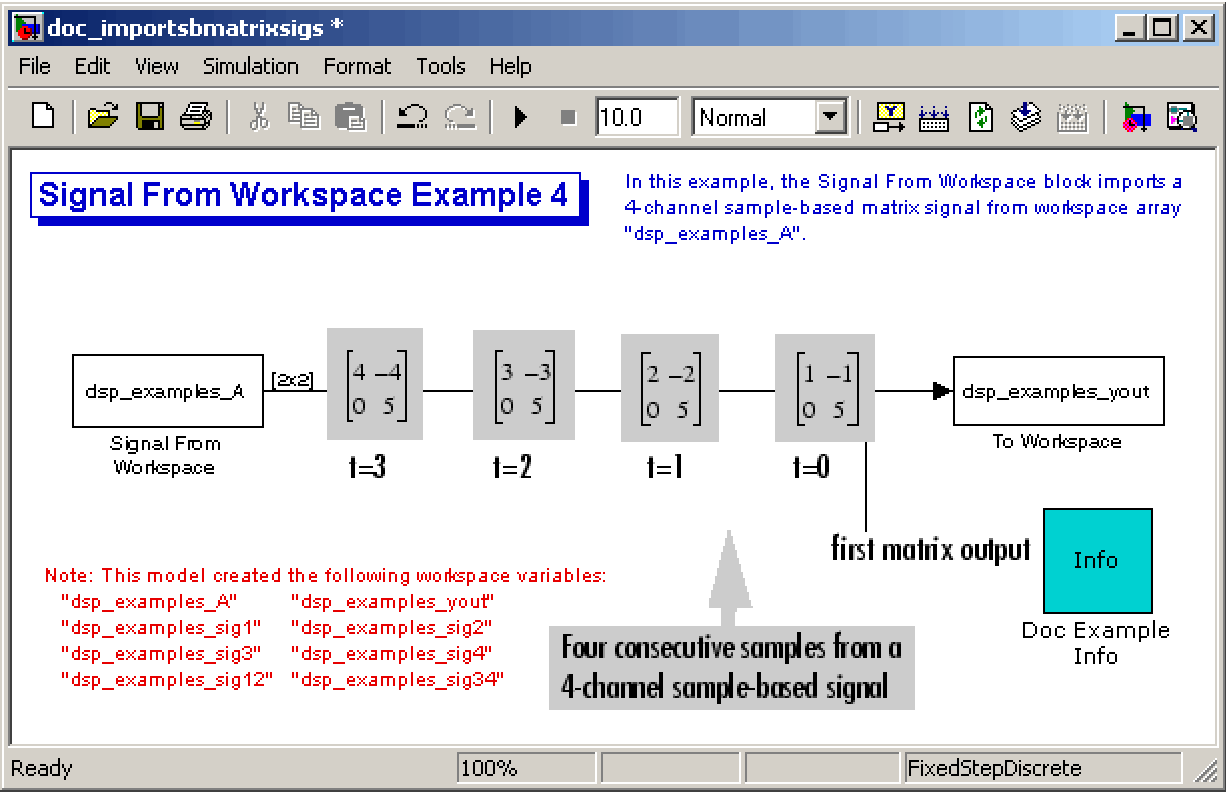
- **Signal** = dsp_examples_A
- **Sample time** = 1
- **Samples per frame** = 1
- **Form output after final data value** = Setting to zero

The dsp_examples_A array represents a four-channel, 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.



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.

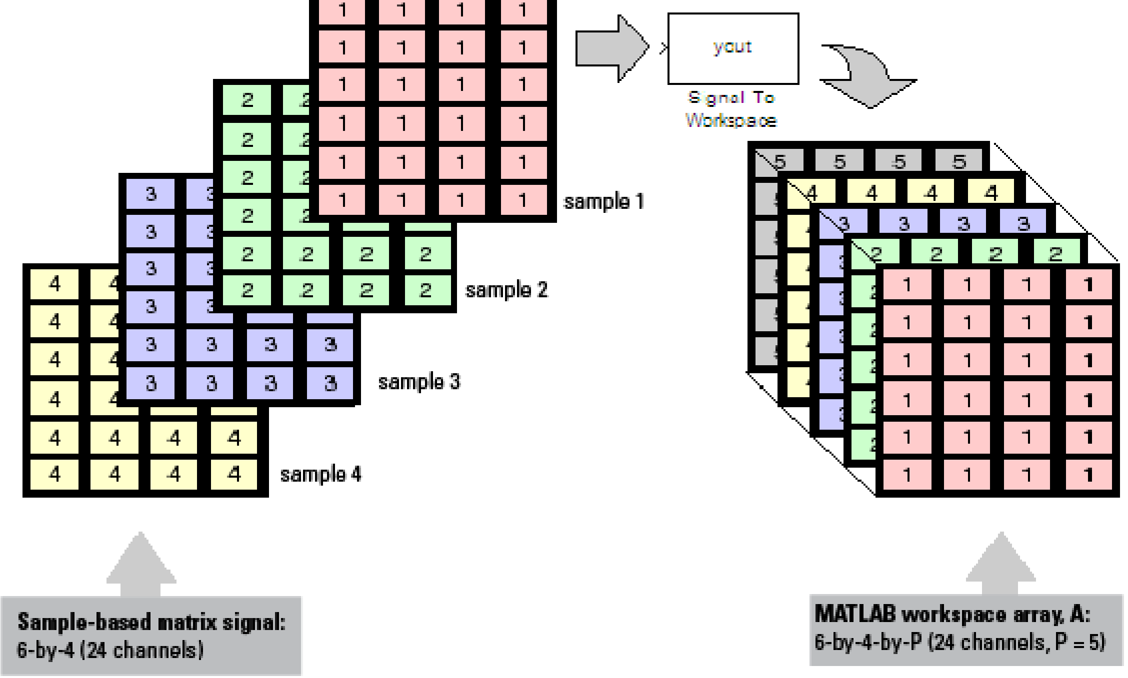
You have now successfully imported a sample-based matrix signal into your model using the Signal From Workspace block.

Exporting 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*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.

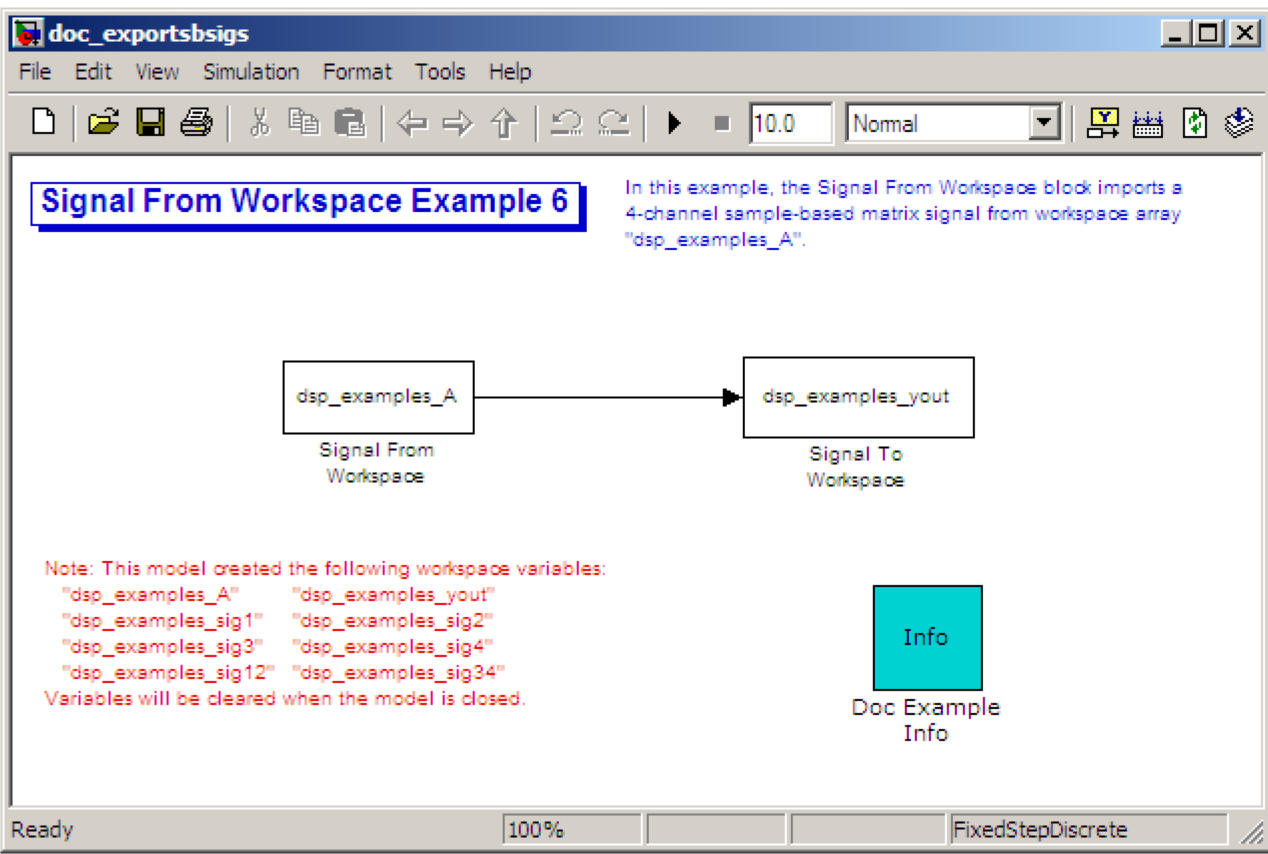
The following figure is the graphical illustration of this process using a 6-by-4 sample-based signal exported to workspace array A.



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 `doc_exportsbsigs` at the MATLAB command line.



Also, the following variables are loaded into the MATLAB workspace:

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

```
dsp_examples_sig34   1x2x100    1600     double array
dsp_examples_sig4    1x1x100    800      double array
```

In this model, the Signal From Workspace block imports a four-channel 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.

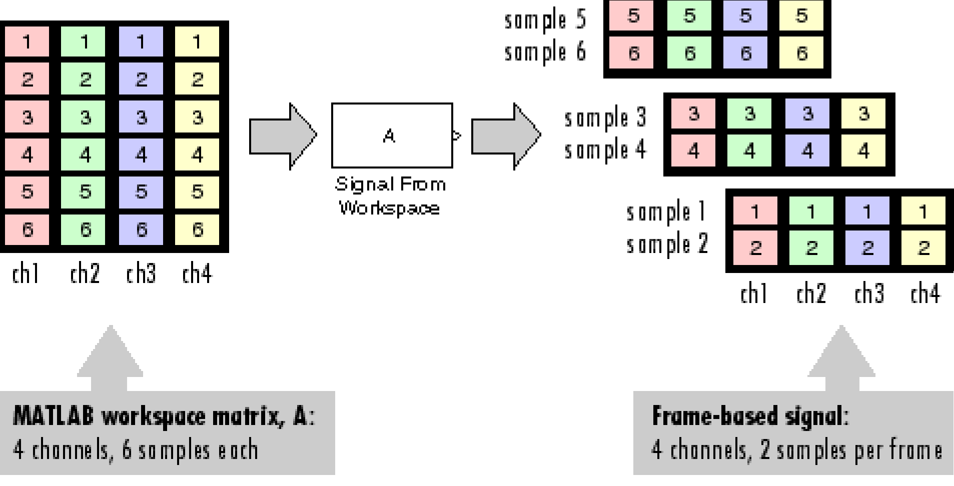
Importing and Exporting Frame-Based Signals

In this section...
"Importing Frame-Based Signals" on page 1-70
"Exporting Frame-Based Signals" on page 1-73

Importing 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 $M*T_s$ 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

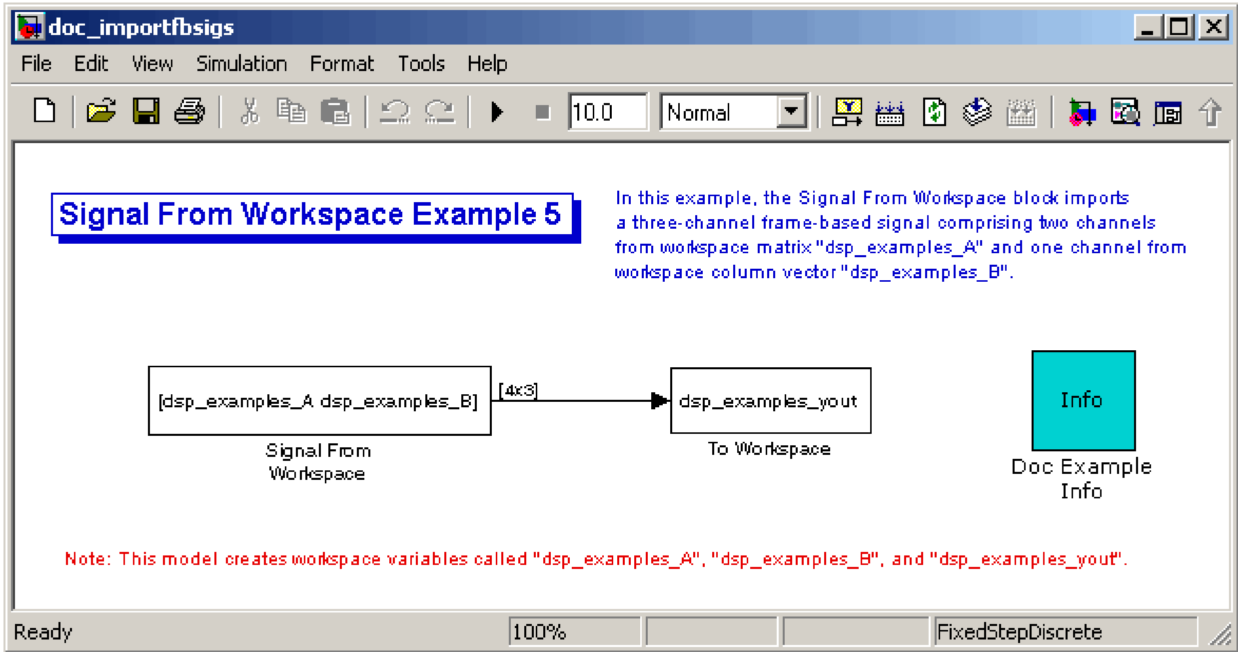
   ```
   doc_importfbsigs
   ```

   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:



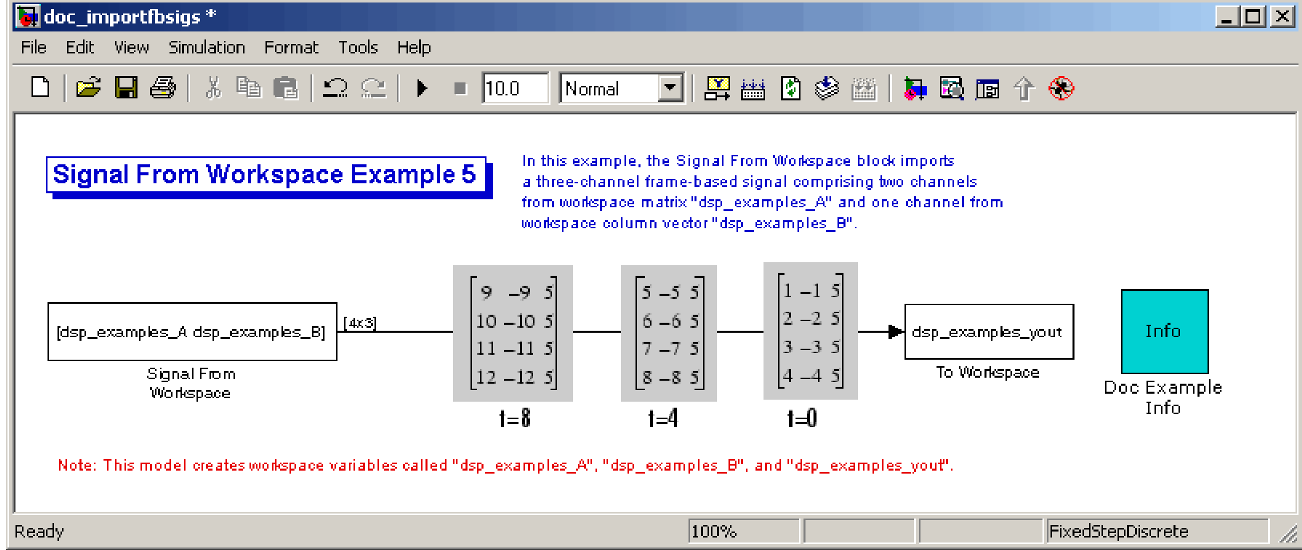
2 Double-click the Signal From Workspace block. Set the block parameters as follows, and then click **OK**:

- **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.



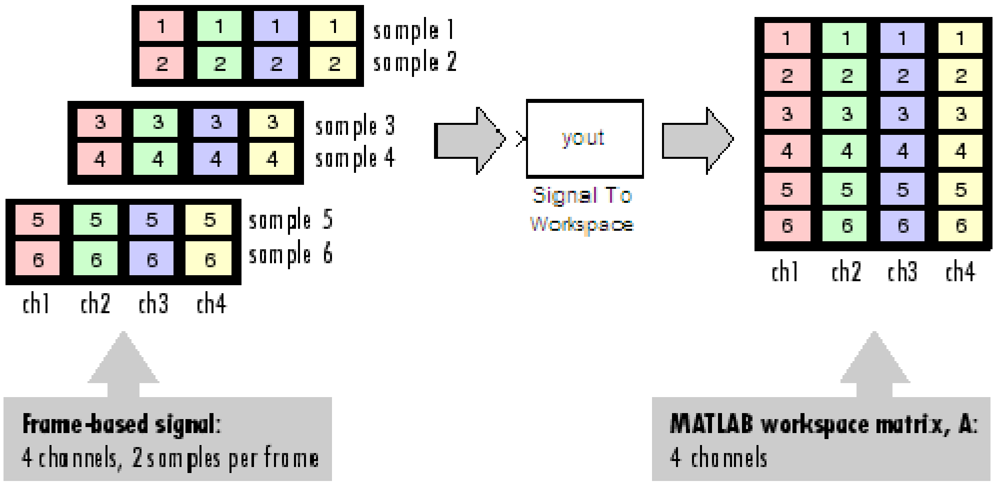
You have now successfully imported a three-channel frame-based signal into your model using the Signal From Workspace block.

Exporting 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.

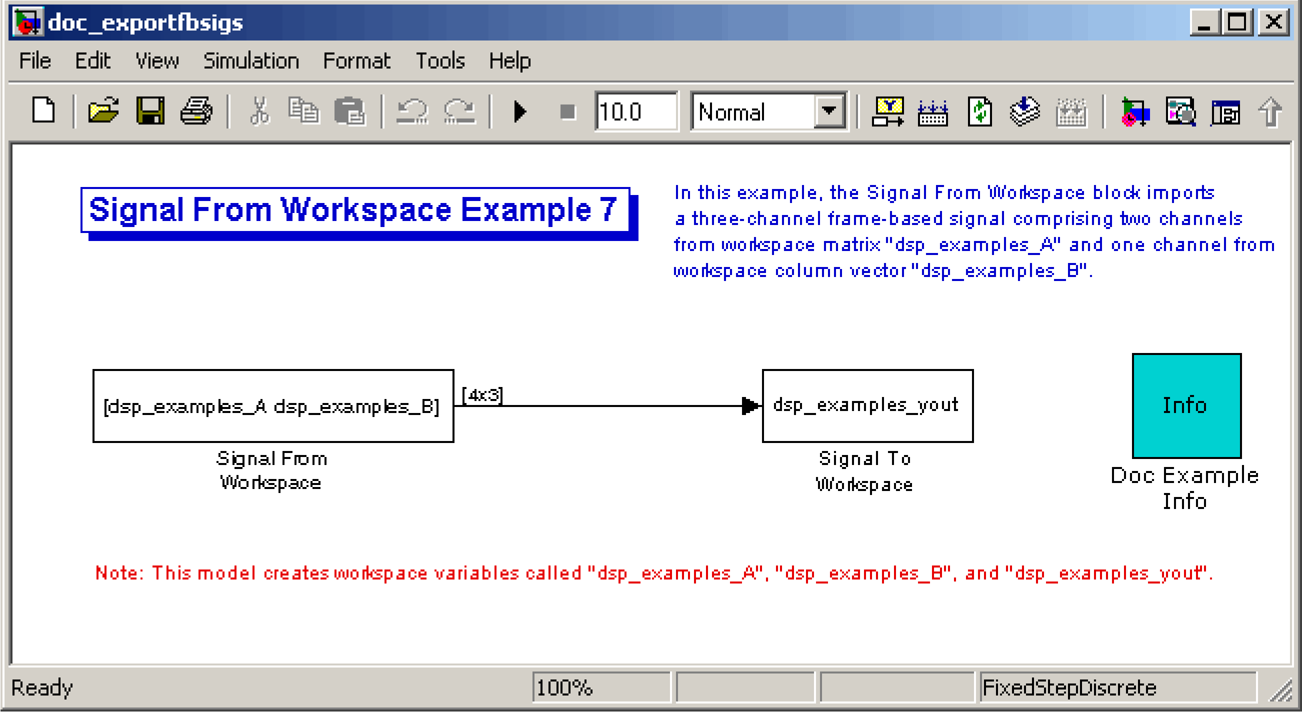
A frame-based signal with N channels and frame size M is represented by a sequence of M-by-N matrices. When the input to the Signal To Workspace block is a frame-based signal, the block creates a P-by-N array in the MATLAB workspace containing the P most recent samples from each channel. The number of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

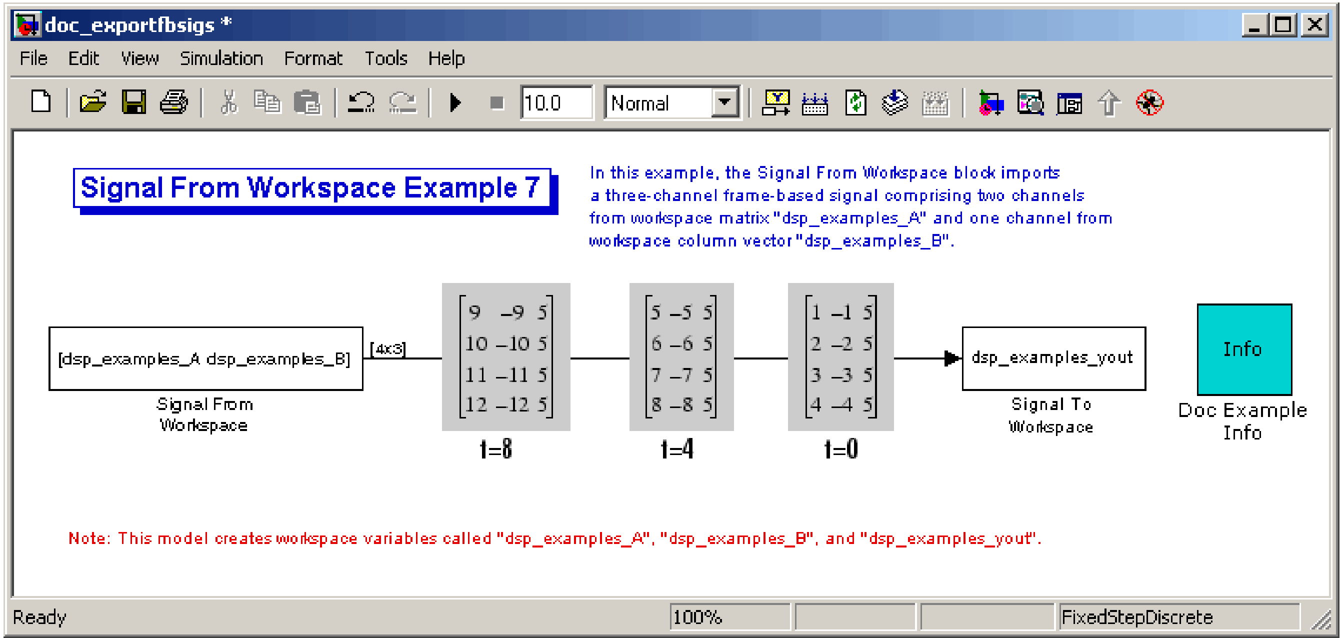
3. Double-click the Signal To Workspace block. Set the block parameters as follows, and then click **OK**:

   - **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.

Displaying Time-Domain Data

You can use Signal Processing Blockset blocks to work with signals in both the time and frequency domain. The Signal Processing Sinks library contains the following blocks for displaying time-domain signals:

- Time Scope
- Vector Scope
- Matrix Viewer
- Waterfall Scope

See the following sections for examples of how you can use the Vector Scope and Time Scope blocks to display time-domain data:

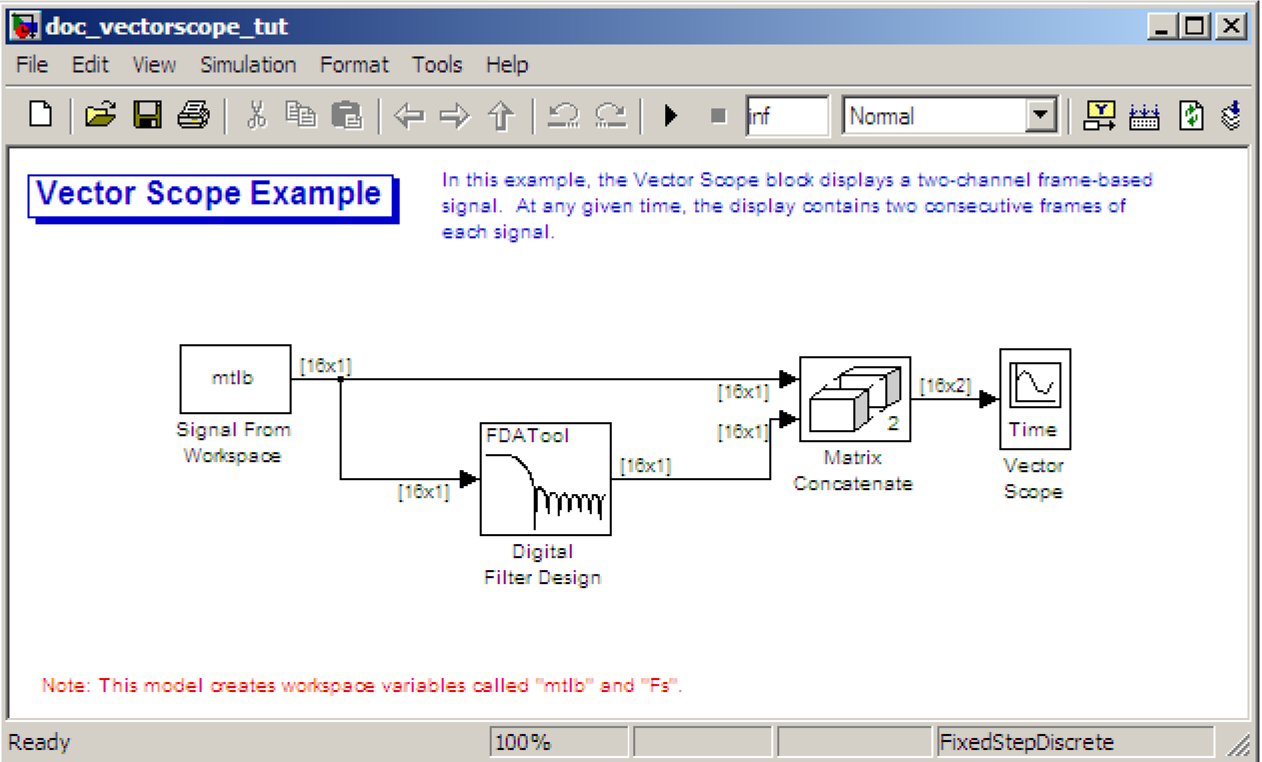
- "Displaying Time Domain Data in the Vector Scope" on page 1-79
- "Displaying Time-Domain Data in the Time Scope" on page 1-82

Displaying Time Domain Data in the Vector Scope

The following example shows you how you can use the Vector Scope block to display time-domain signals:

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

   The Vector Scope Example opens and the variables `Fs` and `mtlb` are loaded into the MATLAB workspace.



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** = `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 outputs a frame-based signal with a frame size of 16 and a sample period of 1 second. The frame period of the signal is 16 seconds. Your input signal is output repeatedly from the Signal From Workspace block.

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

5 Double-click the Digital Filter Design block.

You are going to use this block to filter the input signal in order to produce two distinct signals to send to the Vector Scope block.

6 To specify a lowpass filter, in the **Response Type** section, choose `Lowpass`.

7 In the **Design Method** section, choose `FIR`. Then, from the list, select `Window`.

8 In the **Filter Order** section, select **Specify order** and enter `22`.

9 From the **Window** list, select `Hamming`.

10 In the **Frequency Specifications** section, from the **Units** list, select `Normalized (0 to 1)`.

11 In the **Frequency Specifications** section, set the **wc** parameter to `0.25`.

12 Click **Design Filter**. Then, close the **Block Parameters: Digital Filter Design** dialog box.

13 Double-click the Matrix Concatenate block. The **Function Block Parameters: Matrix Concatenate** dialog box opens.

14 Set the block parameters as follows:

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

Based on these parameters, the Matrix Concatenate block combines the two signals so that each column corresponds to a different signal.

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

16 Double-click the Vector Scope block.

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

- Click the **Scope Properties** tab.
- **Input domain** = `Time`
- **Time display span (number of frames)** = `2`

When you run the model, the Vector Scope block plots two consecutive frames of each channel at each update.

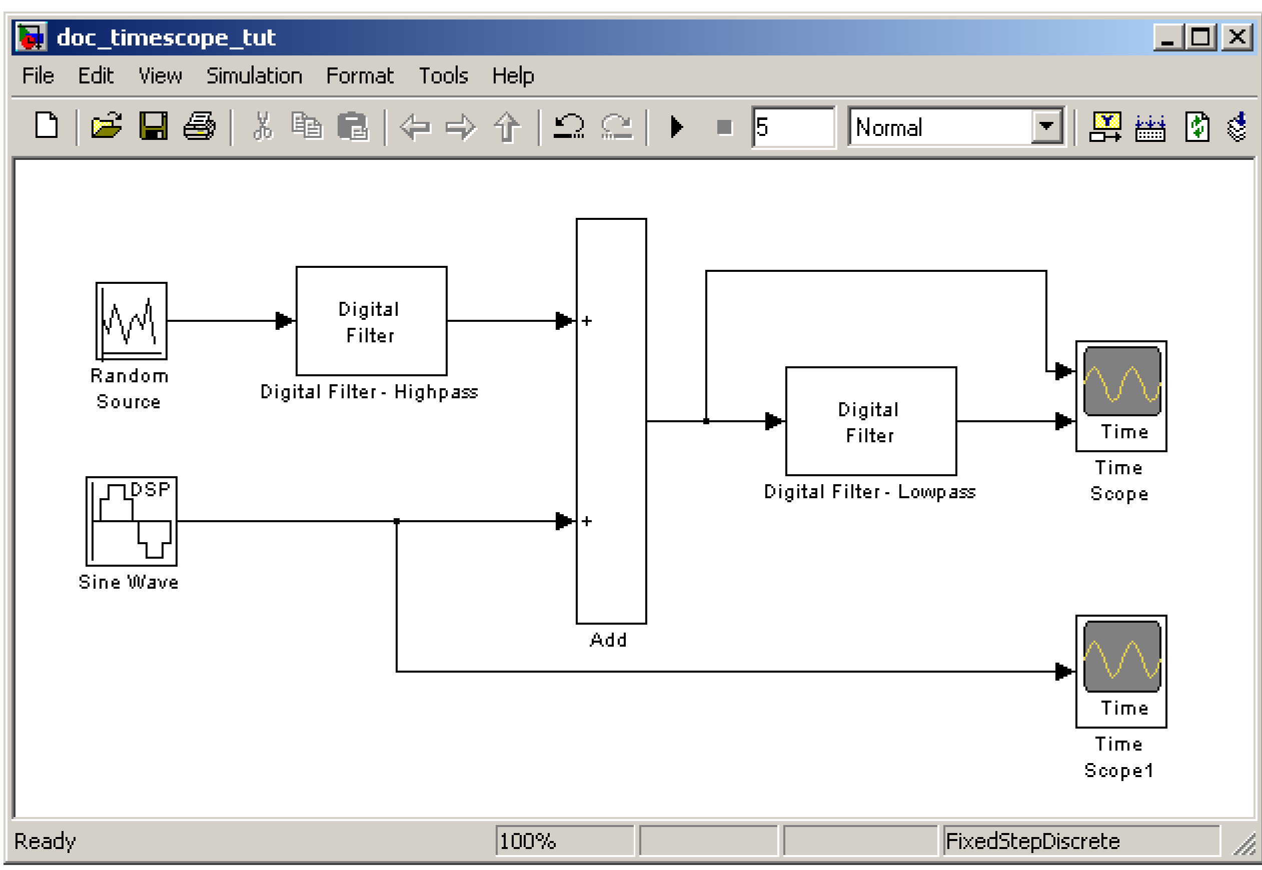
18 Run the model.

The Vector Scope window displays the original signal in blue and the filtered signal in black. To display the channel legend, right-click inside of the Vector Scope window and select **Channel legend** from the menu.

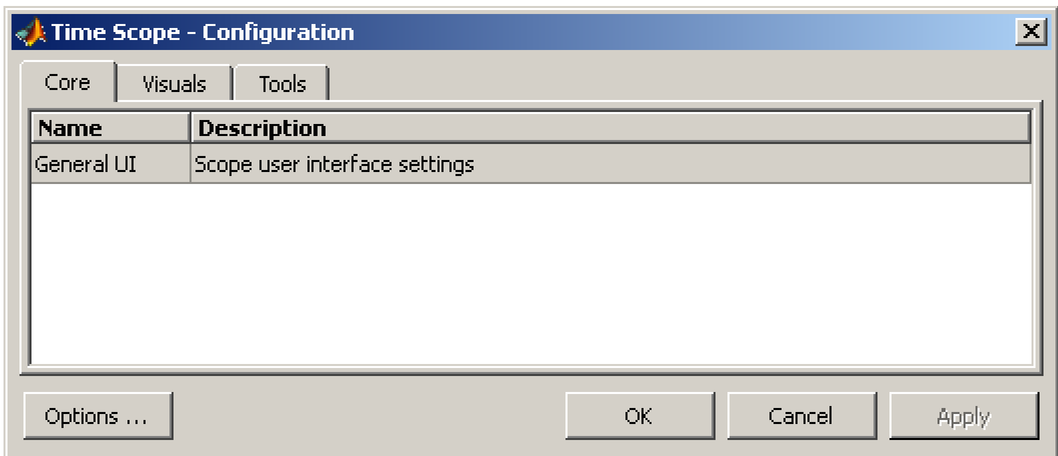
You have now successfully displayed two frame-based signals in the time domain using the Vector Scope block.

Displaying Time-Domain Data in the Time Scope

The following example shows you how to configure the Time Scope blocks in the doc_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.

Example Workflow

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

1. "Configuring the Time Scope" on page 1-85
2. "Using the Playback Controls" on page 1-89
3. "Modifying the Scope Display" on page 1-91
4. "Inspecting Your Data (Scaling the Axes and Zooming)" on page 1-94
5. "Managing Multiple Time Scopes" on page 1-97

To get started with this example, open the model by typing doc_timescope_tut at the MATLAB command line.

Configuring 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 using the **Time Scope – Visuals:Time Domain Options** dialog box.

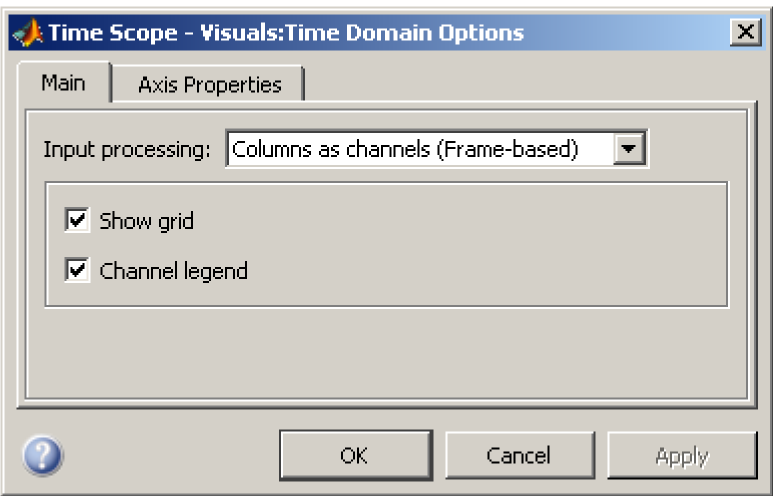
Note As you progress through this workflow, notice the blue question mark icon ( ) in the lower-left corner of the subsequent dialog boxes. This icon indicates that context-sensitive help is available. You can get more information about any of the parameters on the dialog box by right-clicking the parameter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

Configuring Appearance and Specifying Signal Interpretation. To configure the appearance of the Time Scope window and specify how the Time Scope block interprets input signals:

1. Click the **Visuals** tab of the main **Time Scope – Configuration** dialog box.

2. Select **Time Domain**, and click the **Options** button.

   The following options dialog box appears.



This table shows the appropriate parameter settings for the **Main** tab of the **Visuals:Time Domain Options** dialog box.

Parameter	Setting
Input Processing	`Columns as channels (Frame-based)`
Show grid	Checked
Channel legend	Checked

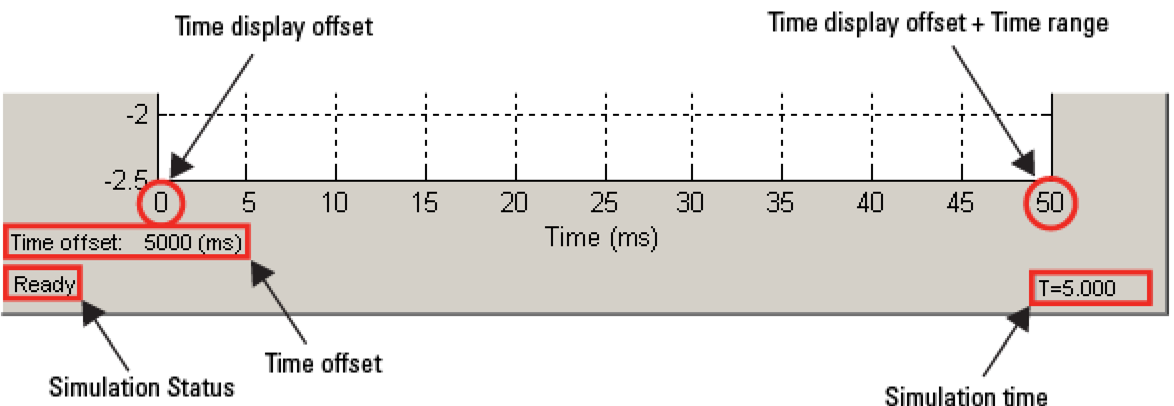
The Time Scope block accepts both sample- and frame-based input signals, but you must specify how the block should handle them. To do so, set the **Input processing** parameter on the **Time Scope – Visuals:Time Domain Options** dialog box to the appropriate choice.

In this example, 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)`.

Setting Axis Properties. Navigate to the **Axis Properties** tab of the **Visuals:Time Domain Options** dialog box, and set the parameters to the values shown in the following table.

Parameter	Setting
Time range	`Input sample time`
Time display offset	`0`
Minimum Y-limit	`-2.5`
Maximum Y-limit	`2.5`
Y-axis label	`Amplitude`

The **Time range** parameter allows you to enter a numeric value, a variable that evaluates to a numeric value, or select the `Input sample time` menu option. The actual range of values that the block displays on the X-axis depends on the value of both the **Time range** and **Time display offset** parameters. See the following figure.

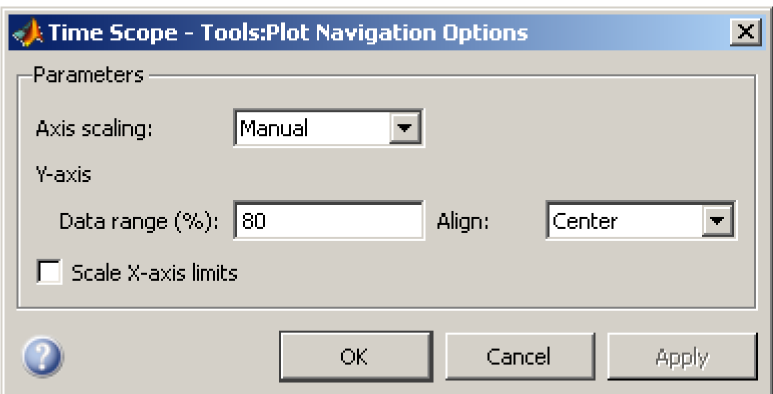


For information on the other labels in the scope window, see the Time Scope reference page. In this example, the values on the X-axis range from `0` to the `Input sample time`, where the `Input sample time` is `0.05` seconds (50 ms). Click **OK** to save your changes and close the **Visuals:Time Domain Options** dialog box.

Configuring 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 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 scope window. You can specify a value for both the Y- and X-axis. The higher the value you enter for the Y-axis **Data range (%)**, the tighter the Y-axis range is with respect to the minimum and maximum values in your signal. For example, to have your signal cover the entire Y-axis range when the block scales the axes, set this value to 100.
Align	Allows you to specify where the block should align your data with respect to each axis. You can choose to have your data aligned with the top, bottom, or center of the Y-axis. Additionally, if you select the **Scale X-axis limits** check box, you can choose to have your data aligned with the right, left, or center of the X-axis.

1 To open the **Plot Navigation options** dialog box, navigate to the **Tools** tab of the main configuration dialog box, and click **Options**.



2 Set the parameters 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 example.

Using the Playback Controls

One of the advantages to using the Time Scope block in your models is that you can control model simulation directly from the scope window. The buttons on the Playback 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.

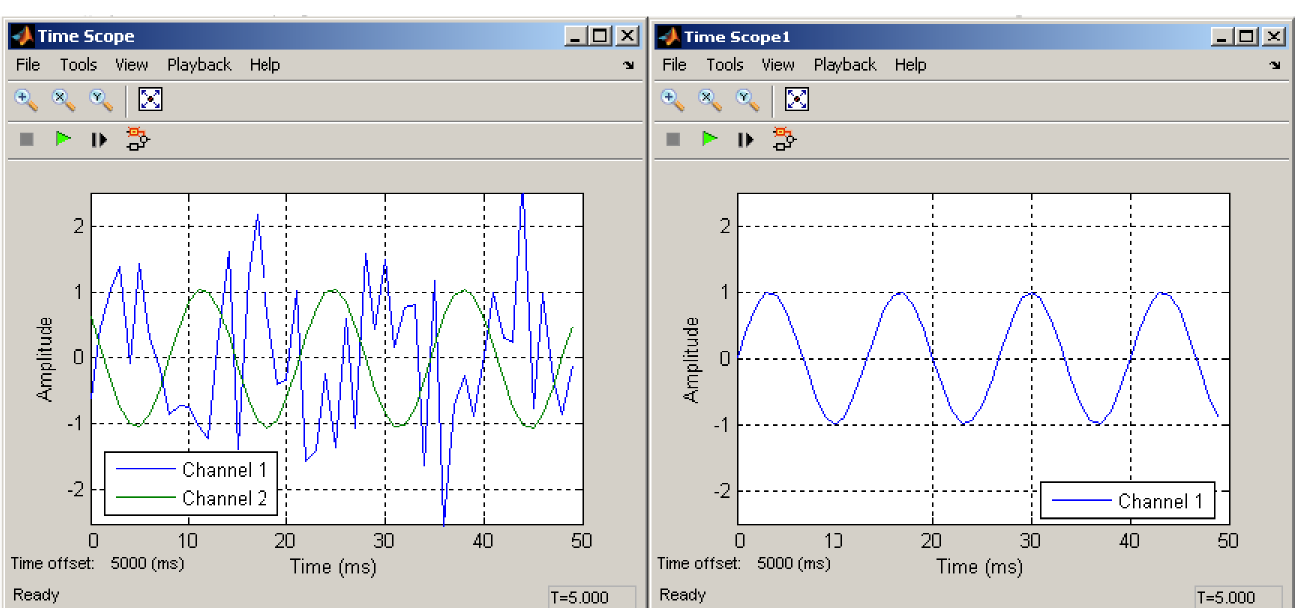
You can access a list of keyboard shortcuts for the Time Scope by selecting **Help > Keyboard Command Help**. The following procedure introduces you to these features.

1. If the Time Scope window is not open, double-click the block icon in the `doc_timescope_tut` model. Start model simulation, by clicking the start button ( ) on the Playback Toolbar of the Time Scope window, or using one of the following keyboard shortcuts:
   - **Ctrl+T**
   - **p**
   - **Space**
2. While the simulation is running and one of the scopes is your active window, pause the simulation by using either of the following keyboard shortcuts:
   - **p**
   - **Space**

   Alternatively, you can pause the simulation by pressing the pause button on the Time Scope window ( ), or by selecting **Playback > Pause** from the scope menu.
3. With the model simulation still paused, advance the simulation by a single time step using the **Simulate one step** button ( ) on the scope window.

   Next, try using keyboard shortcuts to achieve the same result. Press **Page Down** to advance the simulation by one time step, and then use the **Right arrow** key to advance by another time step.
4. Resume model simulation using any of the following methods:
   - Select **Playback > Continue** from the Time Scope menu.
   - Press the **Continue simulation** button ( ) on the Playback Toolbar of the scope window.
   - Use a keyboard shortcut, such as **p** or **Space**.

5 When simulation stops, your scopes should appear as follows.



Modifying the Scope Display

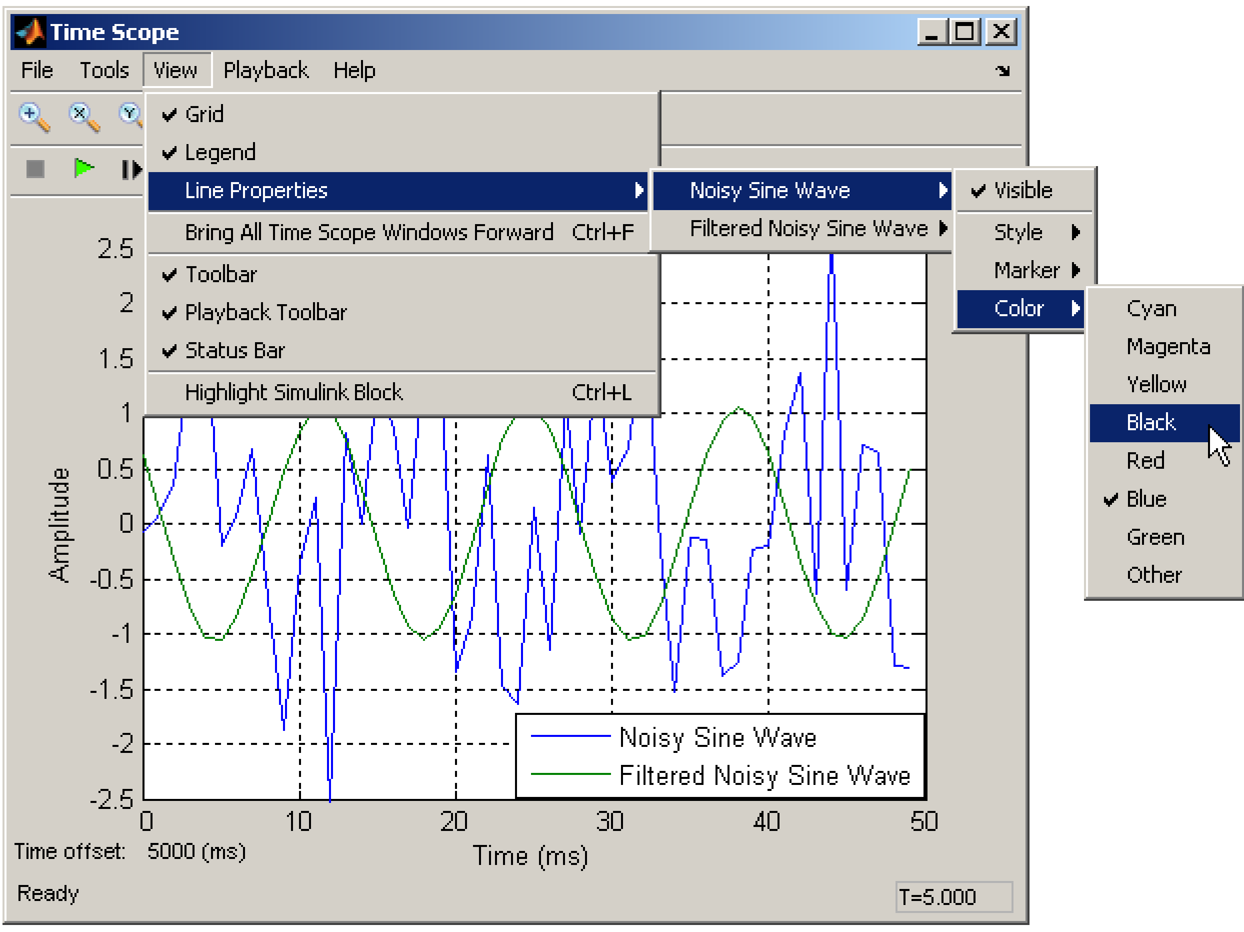
You can control the appearance of the scope window using options from the **View** menu. Among other capabilities, this menu allows you to:

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

Changing 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 scope names the signals Channel 1, Channel 2, etc. For this example, set the signal names as shown in the following table.

Block Name	Original Signal Name	New Signal Name
Time Scope	Channel 1	Noisy Sine Wave
Time Scope	Channel 2	Filtered Noisy Sine Wave
Time Scope1	Channel 1	Original Sine Wave

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



Set the line properties according to the values shown in the following table.

Block Name/Signal Name	Style	Marker	Color
Time Scope/Noisy Sine Wave	—	None	Black
Time Scope/Filtered Noisy Sine Wave	—	diamond	Red
Time Scope1/Original Sine Wave	—	*	Blue

Showing and Hiding 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 playback controls, select **View > Playback Toolbar**. Doing so removes the playback toolbar from the scope window and also removes the check mark from next to the **Playback Toolbar** option in the **View** menu.
- You can choose to show the playback toolbar again at any time by selecting **View > Playback Toolbar**.

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

Inspecting 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 scope window. The axes scaling tools allow you to specify when and how often the scope scales the axes.

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

- Select **Tools > Scale Axes Limits** from the Time Scope menu.

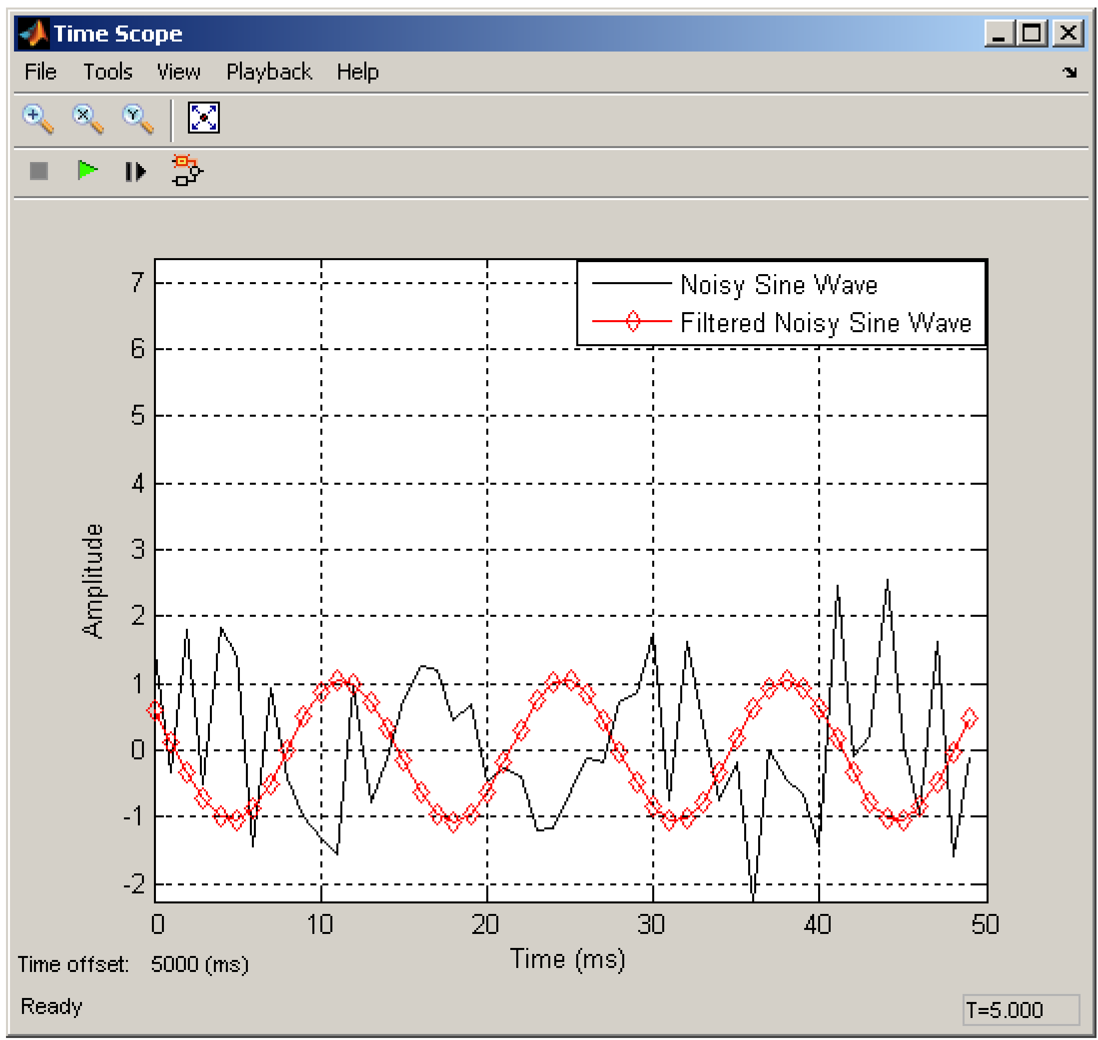
- Press the **Scale Axes Limits** toolbar button ( ).
- With the scope as your active window, press **Ctrl + A**.

Adjusting 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 using the **Data range (%)** and **Align** parameters. In a previous section, you set these parameters to `80` and `Center`, respectively.

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. You can do so from the Tools tab of the main Configuration dialog box, or by selecting **Tools > Axes Scaling Options** from the scope menu.

- 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.



Using the Zoom Tools. The zoom tools allow you to zoom in simultaneously in both the X and Y directions, 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 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, click and drag your cursor from the 10 ms mark to the 20 ms mark on the scope window.
- To zoom out of the 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 scope window and selecting **Reset to Original View**.

Managing 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 example, `doc_timescope_tut`, contains two Time Scope blocks; `Time Scope` and `Time Scope1`. The following sections discuss the tools you can use to manage these Time Scope blocks.

Opening 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, this menu option opens the window and brings it into view.

To try this menu option in the `doc_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 and comes into view, along with the already active `Time Scope` window.

Opening 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 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 scope window 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.

Finding 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 which signals they are attached to.

To try this feature, open the `Time Scope` window, and click the **Highlight Simulink Block** button on the playback toolbar. Doing so brings the `doc_timescope_tut` model into view. The `Time Scope` block flashes three times in the model window, allowing you to see where in your model the block is located.

Closing 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 `doc_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 scope windows at simulation start by selecting **File > Open at Start of Simulation** from the Time Scope window.

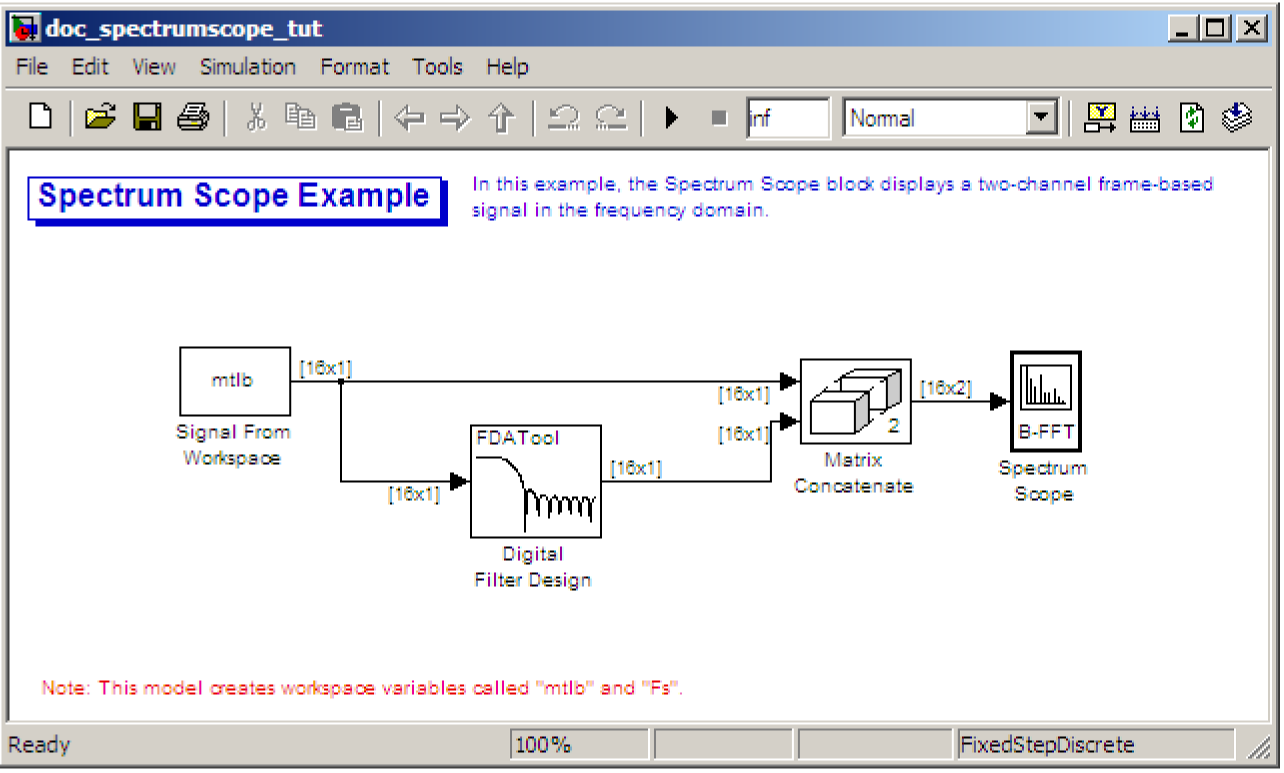
Displaying Frequency-Domain Data

You can use Signal Processing Blockset blocks to work with signals in both the time and frequency domain. To display frequency-domain signals, you can use blocks from the Signal Processing 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 `doc_spectrumscope_tut`.

   The Spectrum Scope Example opens.



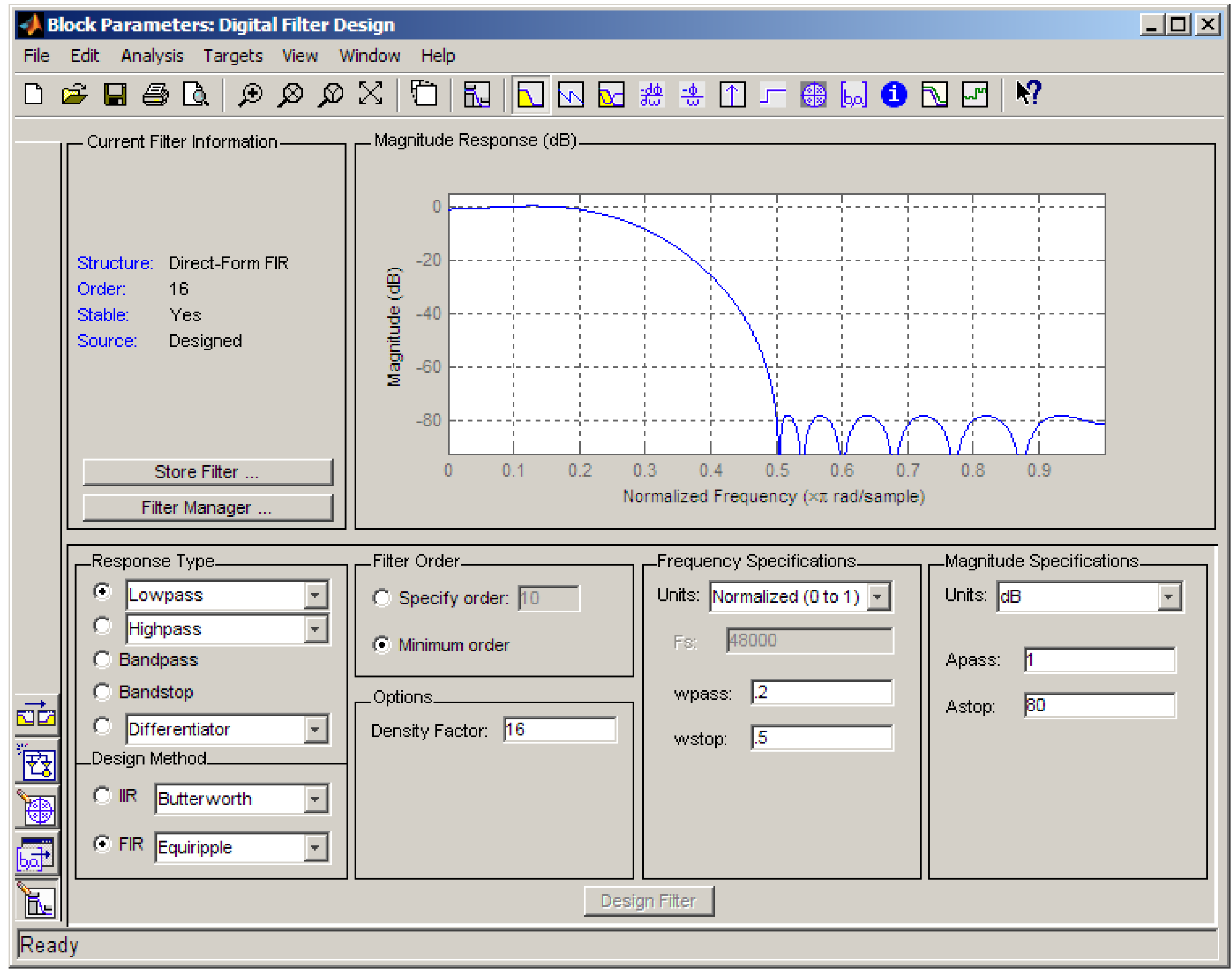
Also, the variables `Fs` and `mtlb` are loaded into the MATLAB workspace.

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.

3 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.



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

- **Number of inputs** = 2

- **Mode** = `Multidimensional array`
- **Concatenate dimension** = `2`

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

5 Double-click the Spectrum Scope block. On the **Scope Properties** tab, set the block parameters as follows, and then click **OK**:

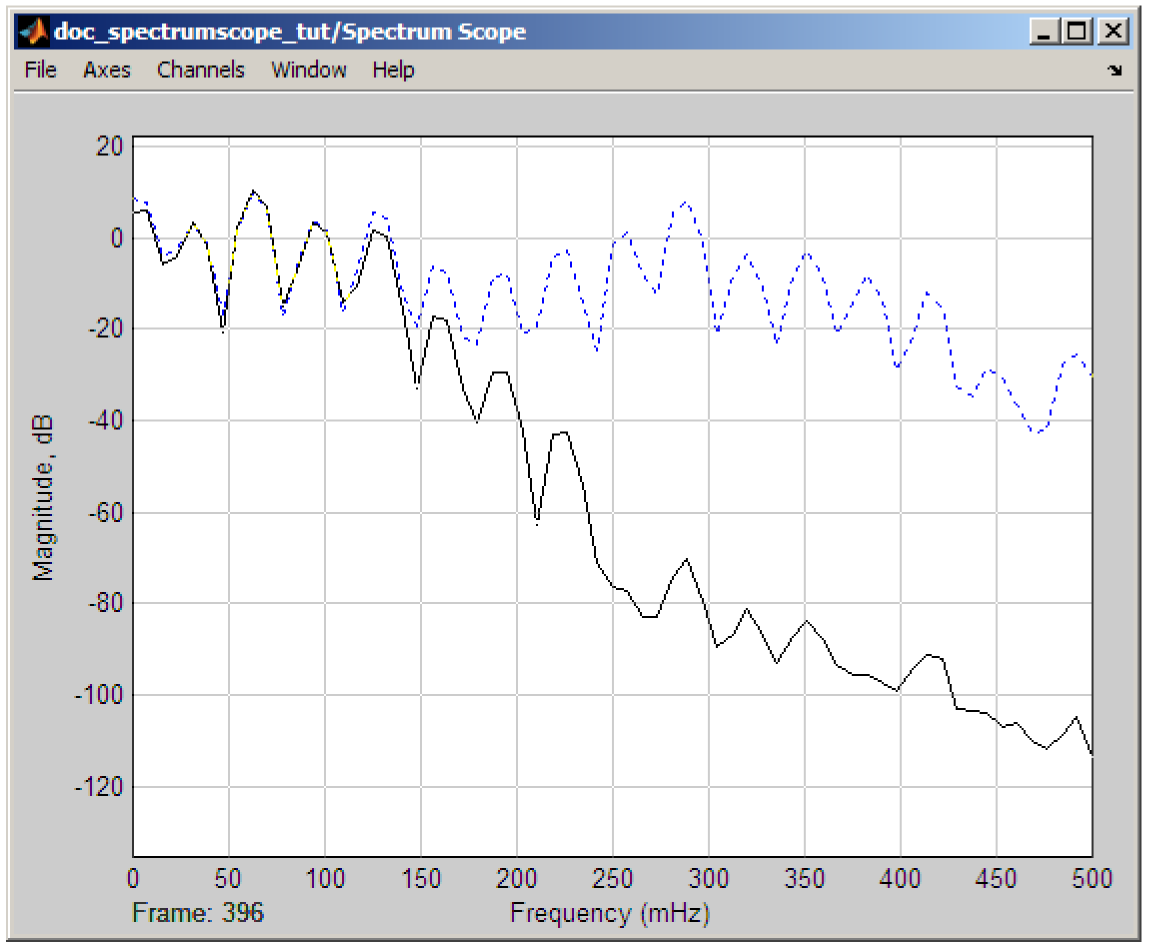
- Select the **Buffer input** check box.
- **Buffer size** = `128`
- **Buffer overlap** = `64`
- **Window type** = `Hann`
- **Window sampling** = `Periodic`
- Clear the **Specify FFT length** check box.
- **Number of spectral averages** = `2`

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 Scope block to display two, frame-based signals in the frequency domain.

Advanced Signal Concepts

This chapter helps you understand how to inspect and convert sample and frame rates. It also explains how to change a sample-based signal into a frame-based signal. Finally, it discusses the concept of delay and describes how this delay can be minimized.

- “Inspecting Sample Rates and Frame Rates” on page 2-2
- “Converting Sample and Frame Rates” on page 2-11
- “Converting Frame Status” on page 2-33
- “Delay and Latency” on page 2-49

Inspecting Sample Rates and Frame Rates

In this section...

"Sample Rate and Frame Rate Concepts" on page 2-2

"Inspecting Sample-Based Signals Using the Probe Block" on page 2-3

"Inspecting Frame-Based Signals Using the Probe Block" on page 2-5

"Inspecting Sample-Based Signals Using Color Coding" on page 2-7

"Inspecting Frame-Based Signals Using Color Coding" on page 2-9

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:

$$T_s = T_f / M$$

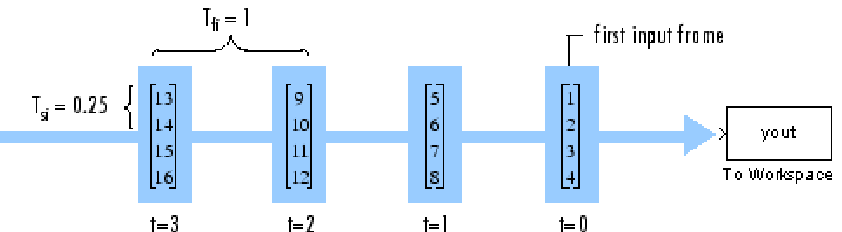
More specifically, the sample periods of inputs (T_{si}) and outputs (T_{so}) are related to their respective frame periods by

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

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

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.

Inspecting Sample-Based Signals Using the Probe Block

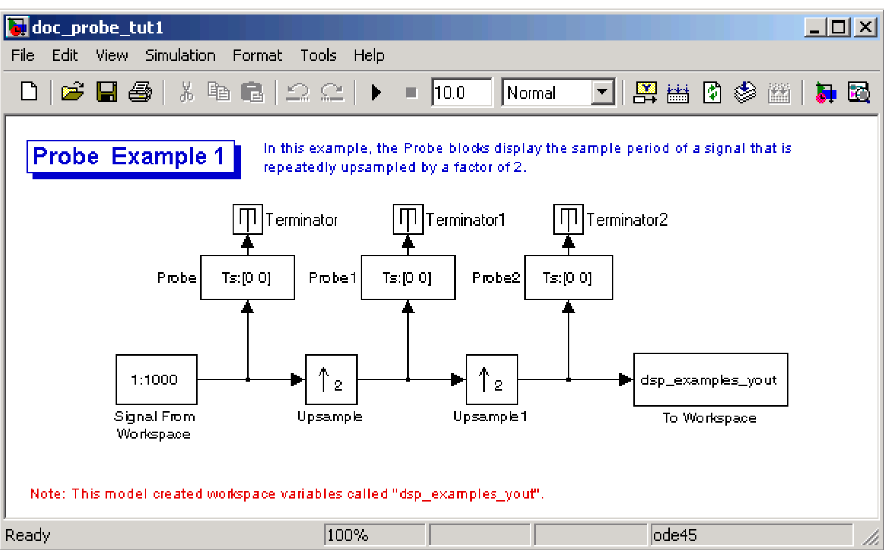
You can use the Probe block to display the sample period of a sample-based signal. For sample-based signals, the Probe block displays the label `Ts`, the sample period of the sequence, followed by a two-element vector. The left element is the period of the signal being measured. The right element is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

Note Simulink offers the ability to shift 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 Signal Processing Blockset blocks do not support them.

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

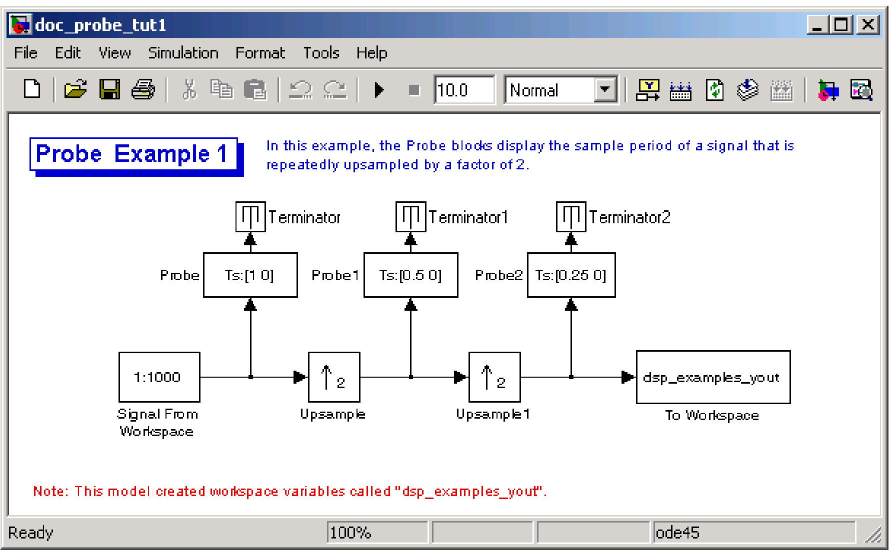
1 At the MATLAB command prompt, type `doc_probe_tut1`.

   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.

Inspecting Frame-Based Signals Using the Probe Block

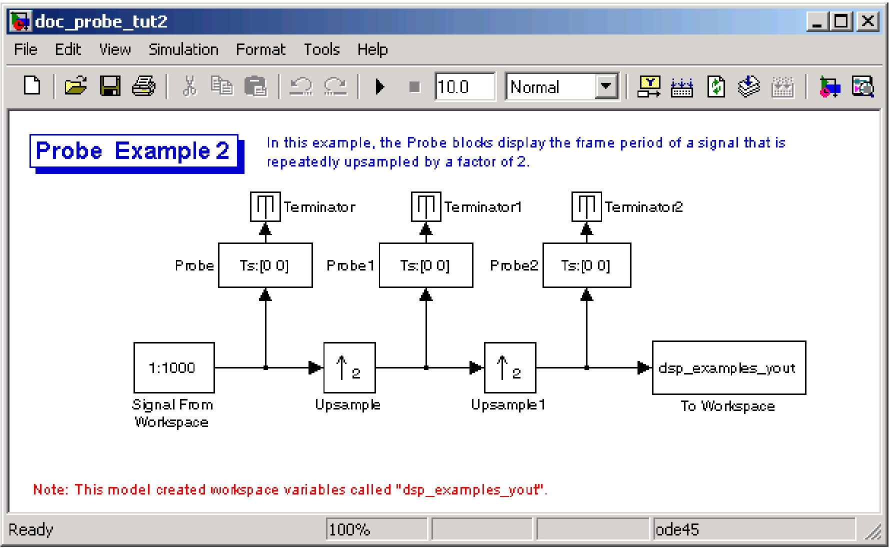
You can use the Probe block to display the frame period of a frame-based signal. For frame-based signals, the block displays the label `Tf`, the frame period of the sequence, followed by a two-element vector. The left element is the period of the signal being measured. The right element is the signal's 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 Signal Processing Blockset blocks do not support them.

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

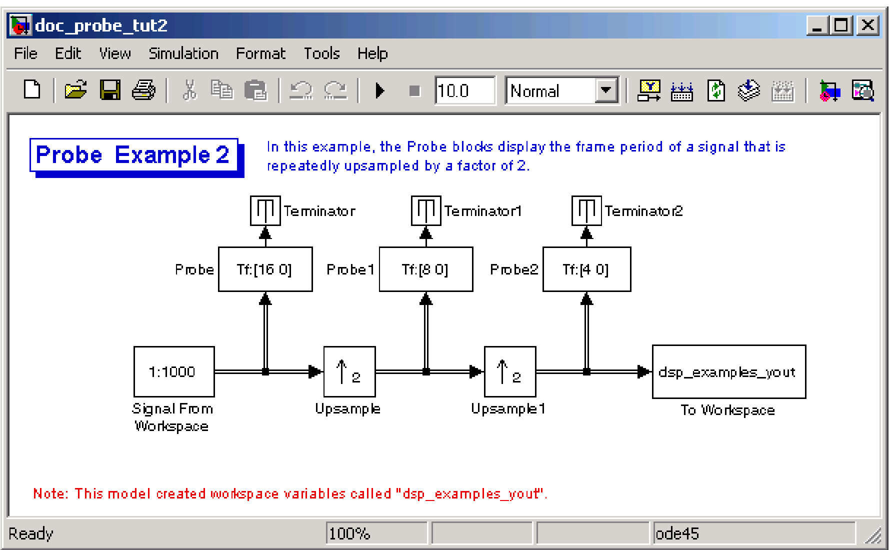
1 At the MATLAB command prompt, type `doc_probe_tut2`.

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.

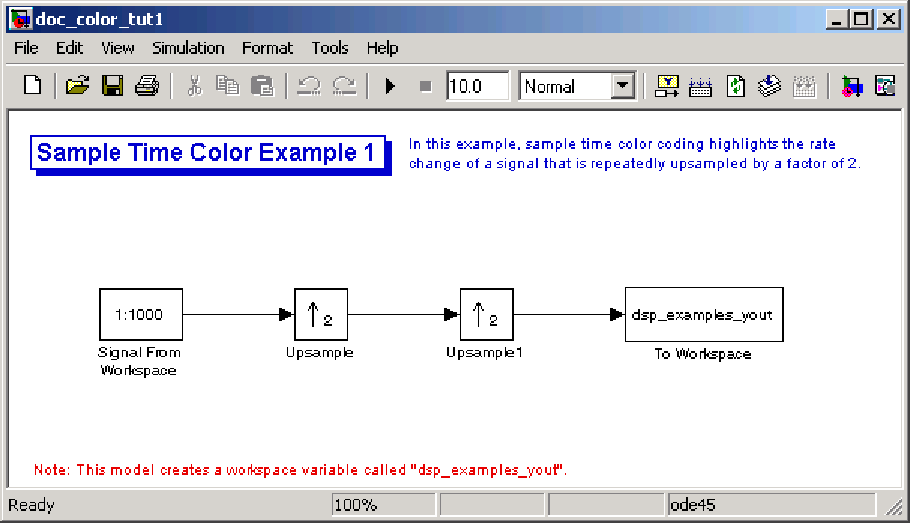
Note that the sample rate conversion is implemented through a change in the frame period rather than the frame size. This is because the **Frame-based mode** parameter in the Upsample blocks is set to `Maintain input frame size` rather than `Maintain input frame rate`.

Inspecting 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 `doc_color_tut1`.

The Sample Time Color Example 1 model opens.

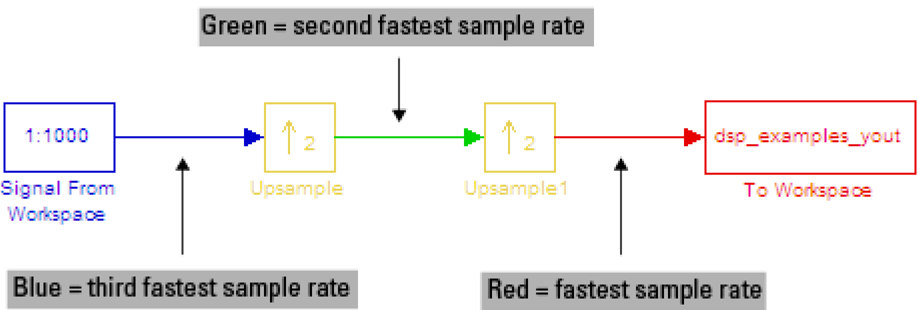


2 From the **Format** menu, point to **Sample Time Display**, and select **Colors**.

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.

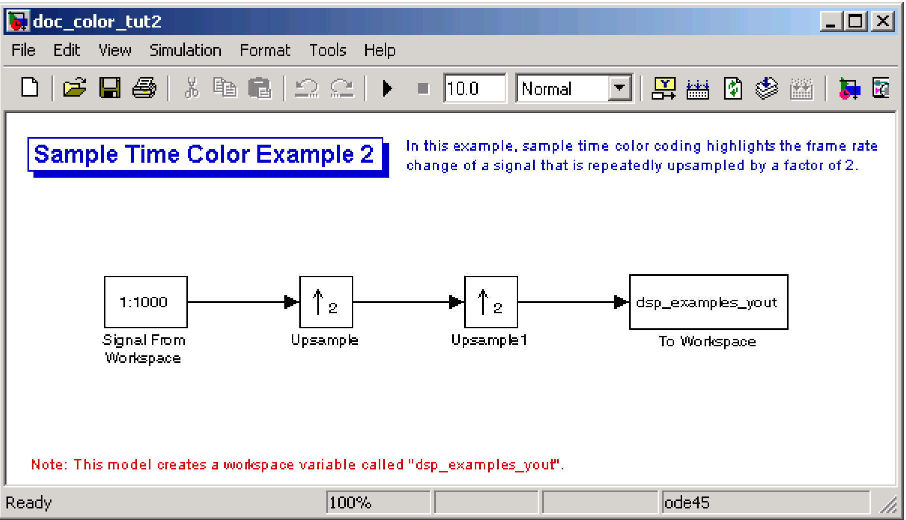
For more information about sample time color coding, see "How to View Sample Time Information" in the Simulink documentation.

Inspecting 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 `doc_color_tut2`.

   The Sample Time Color Example 2 model opens.

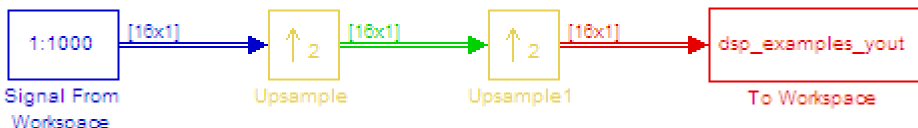


2 To turn on sample time color coding, from the **Format** menu, point to **Sample Time Display**, and select **Colors**.

   Simulink now assigns each frame rate a different color.

3 Run the model.

The model should now look similar to the following figure:

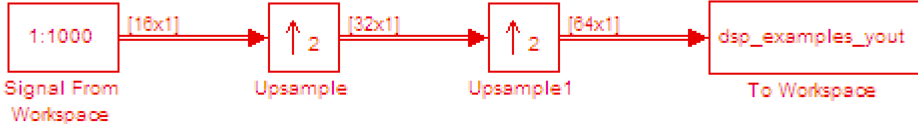


Because the **Frame-based mode** parameter in the Upsample blocks is set to `Maintain input frame size` rather than `Maintain input frame rate`, each Upsample block changes the frame rate. Therefore, each frame-based signal in the model is assigned a different color.

4 Double-click on each Upsample block and change the **Frame-based mode** parameter to `Maintain input frame rate`.

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.

Converting Sample and Frame Rates

In this section...

"Rate Conversion Blocks" on page 2-11

"Rate Conversion by Frame-Rate Adjustment" on page 2-12

"Rate Conversion by Frame-Size Adjustment" on page 2-15

"Avoiding Unintended Rate Conversion" on page 2-19

"Frame Rebuffering Blocks" on page 2-24

"Buffering with Preservation of the Signal" on page 2-27

"Buffering with Alteration of the Signal" on page 2-30

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 Signal Processing Blockset software. Blocks marked with an asterisk (*) offer the option of changing the rate by either adjusting the frame size or frame rate.

Block	Library
Downsample *	Signal Operations
Dyadic Analysis Filter Bank	Filtering / Multirate Filters
Dyadic Synthesis Filter Bank	Filtering / Multirate Filters
FIR Decimation *	Filtering / Multirate Filters
FIR Interpolation *	Filtering / Multirate Filters
FIR Rate Conversion	Filtering / Multirate Filters

Block	Library
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 **Frame-based mode** parameter offering two options for adjusting the sample rate of the signal:

- `Maintain input frame rate`: Change the sample rate by changing the frame size (that is, $M_o \neq M_i$), but keep the frame rate constant ($T_{fo} = T_{fi}$).
- `Maintain input frame size`: Change the sample rate by changing the output frame rate (that is $T_{fo} \neq T_{fi}$), but keep the frame size constant ($M_o = M_i$).

The setting of this parameter does not affect sample-based inputs.

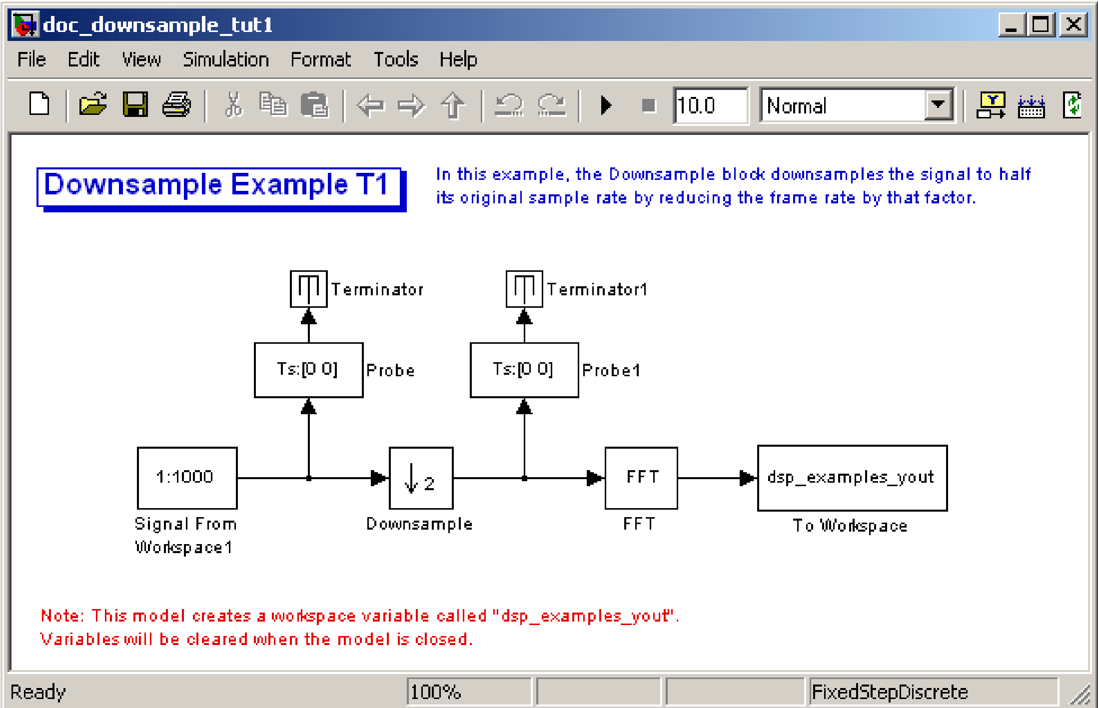
Note When a Simulink model contains signals with various frame rates, the model is called *multirate*. You can find a discussion of multirate models in "Excess Algorithmic Delay (Tasking Latency)" on page 2-57. Also see "Scheduling Considerations" in the Real-Time Workshop® documentation.

Rate Conversion by Frame-Rate Adjustment

One way to change the sample rate of a signal, $1/T_{so}$, is to change the output frame rate ($T_{fo} \neq T_{fi}$), while keeping the frame size constant ($M_o = M_i$). Note that the sample rate of a signal is defined as $1/T_{so} = M_o/T_{fo}$:

1. At the MATLAB command prompt, type `doc_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 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**.

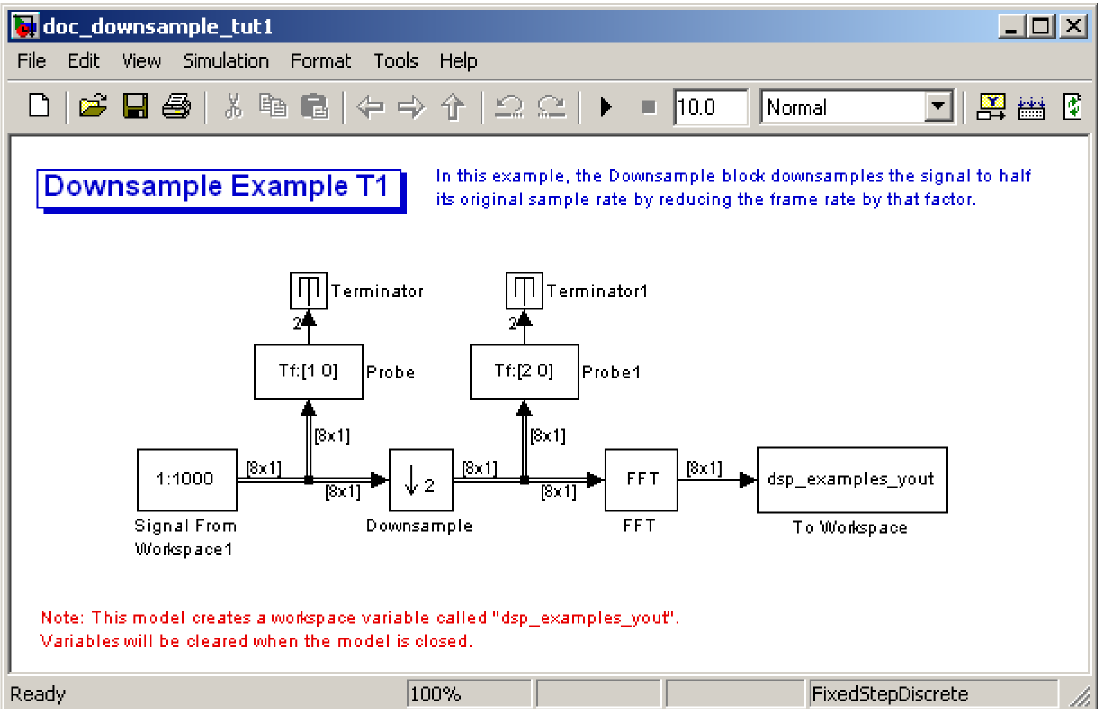
6 Double-click the Downsample block. The **Block Parameters: Downsample** dialog box opens.

7 Set the **Frame-based mode** parameter to `Maintain input frame size`, 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.

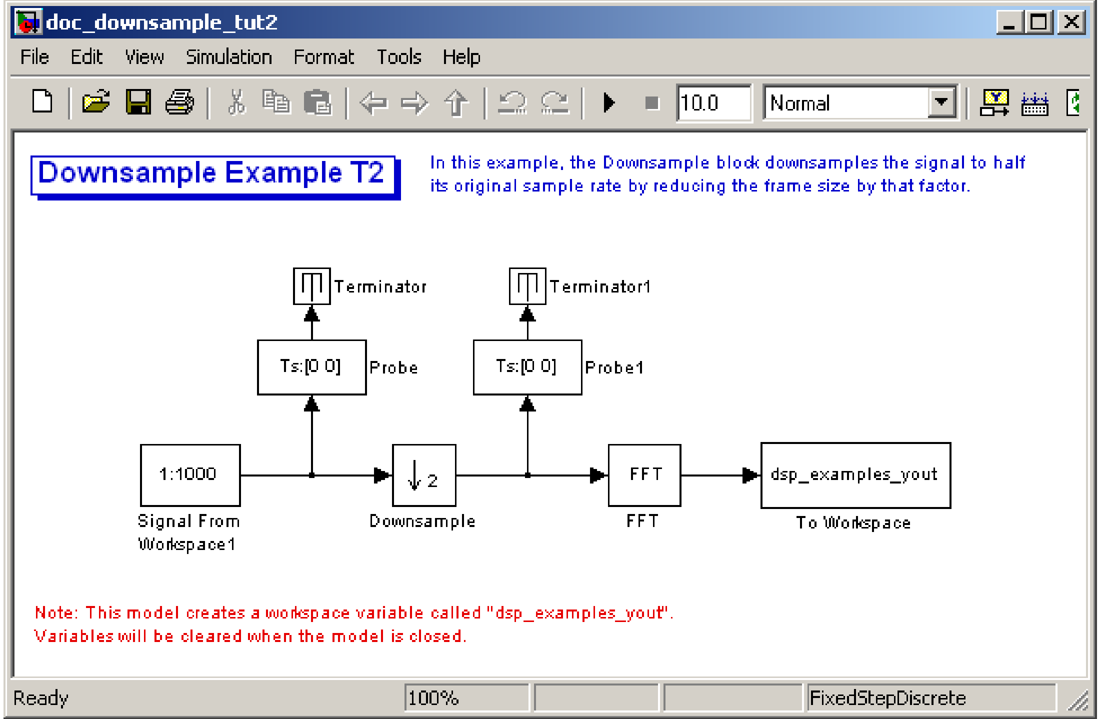
The second Probe block in the model verifies that the output from the Downsample block has a frame period, T_{fo}, of 2 seconds, twice the frame period of the input. However, because the frame rate of the output, $1/T_{fo}$, is 0.5 second, the Downsample block actually downsampled the original signal 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 `doc_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**.

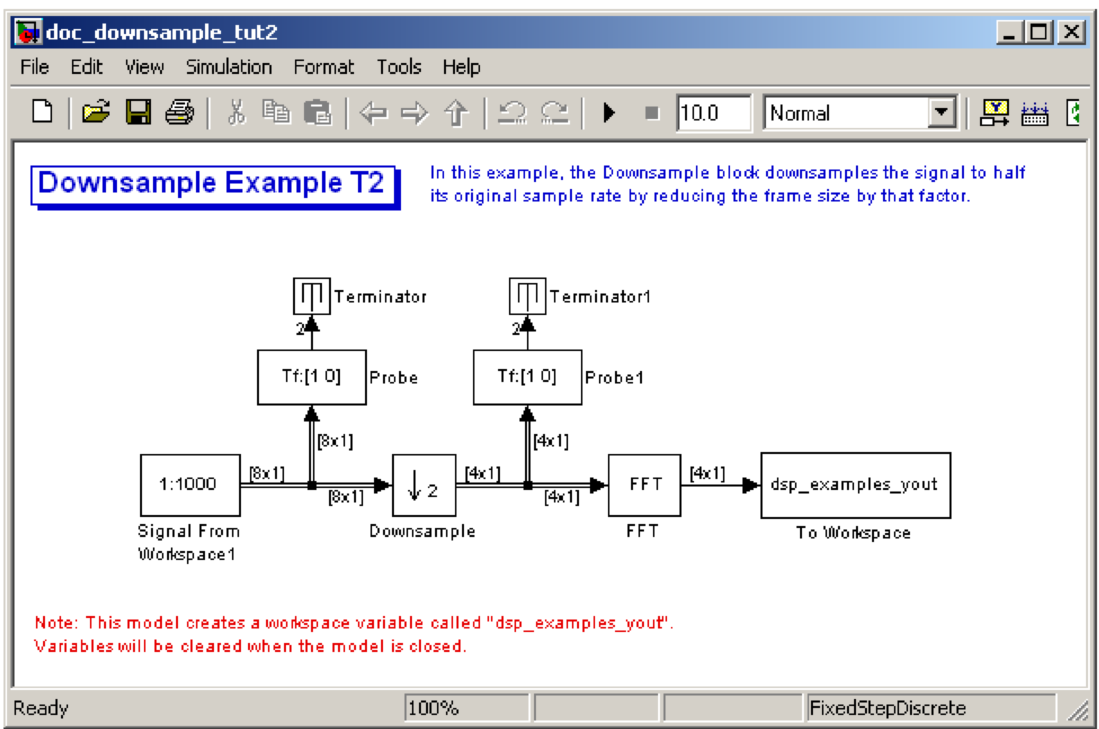
6. Double-click the Downsample block. The **Block Parameters: Downsample** dialog box opens.

7. Set the **Frame-based mode** parameter to `Maintain input frame rate`, 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 ($T_{fi} = T_{fo}$).

Avoiding 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 `doc_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*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).

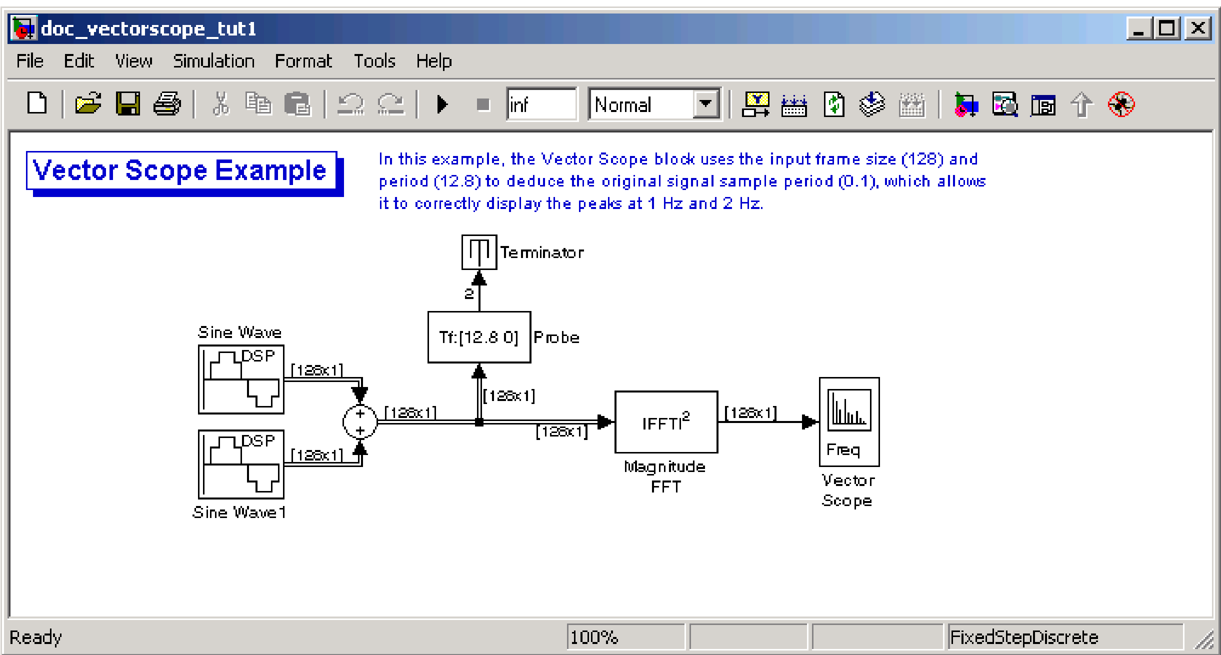
9 Double-click the Vector Scope block.

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

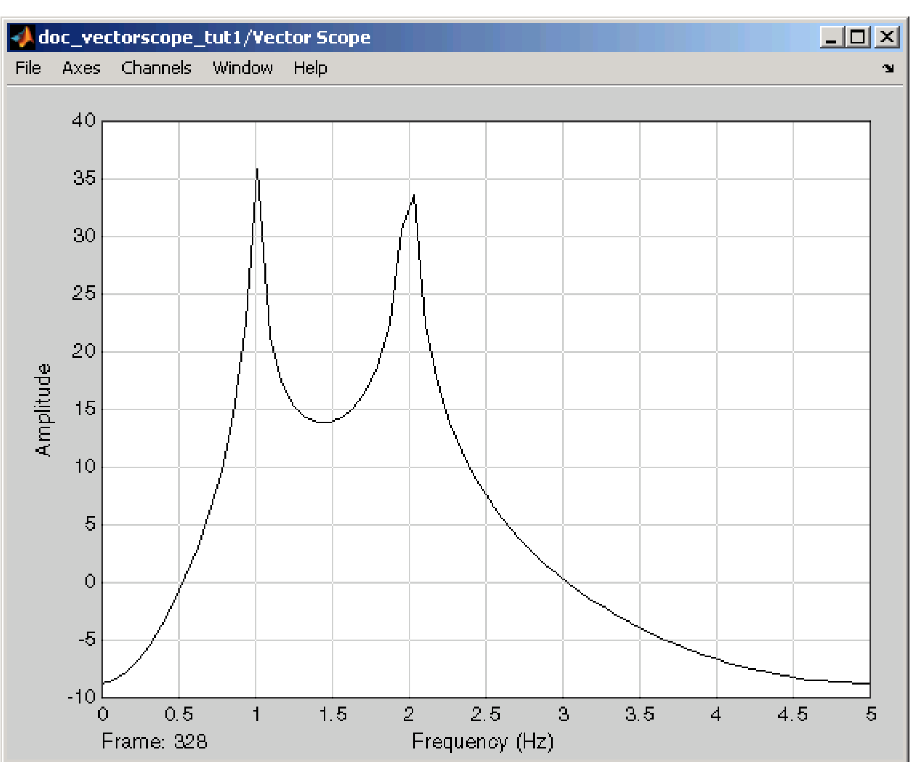
- Click the **Scope Properties** tab.
- **Input domain** = `Frequency`
- Click the **Axis Properties** tab.
- **Minimum Y-limit** = `-10`
- **Maximum Y-limit** = `40`

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.

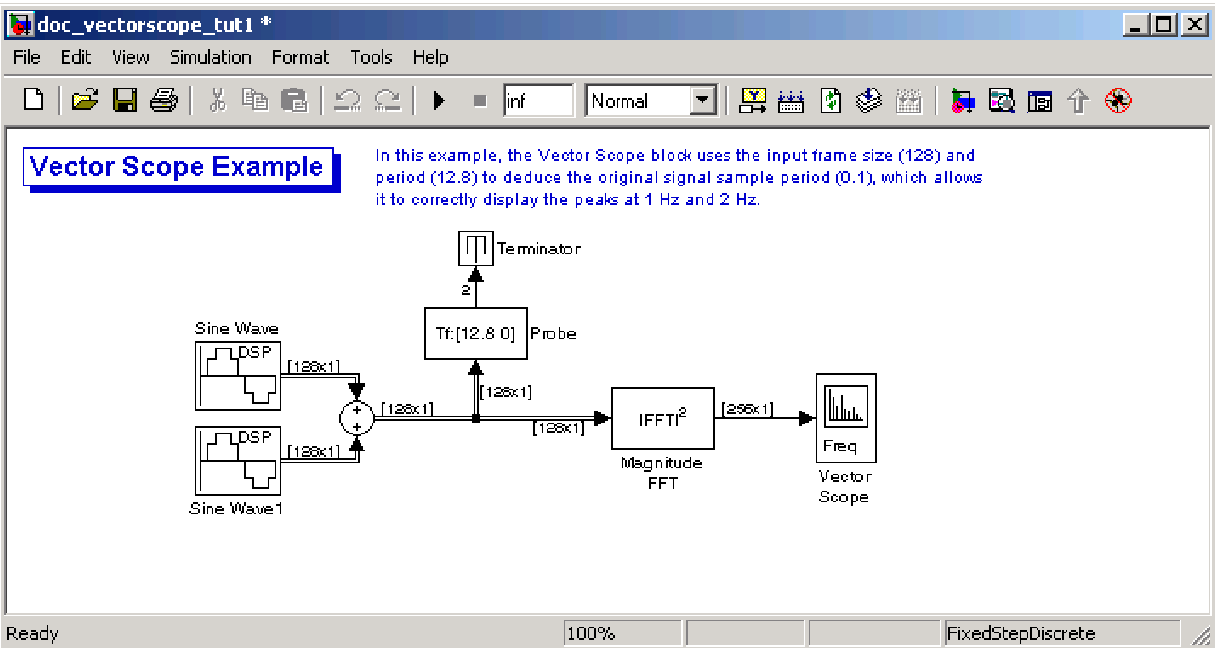
13 Set the block parameters as follows:

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

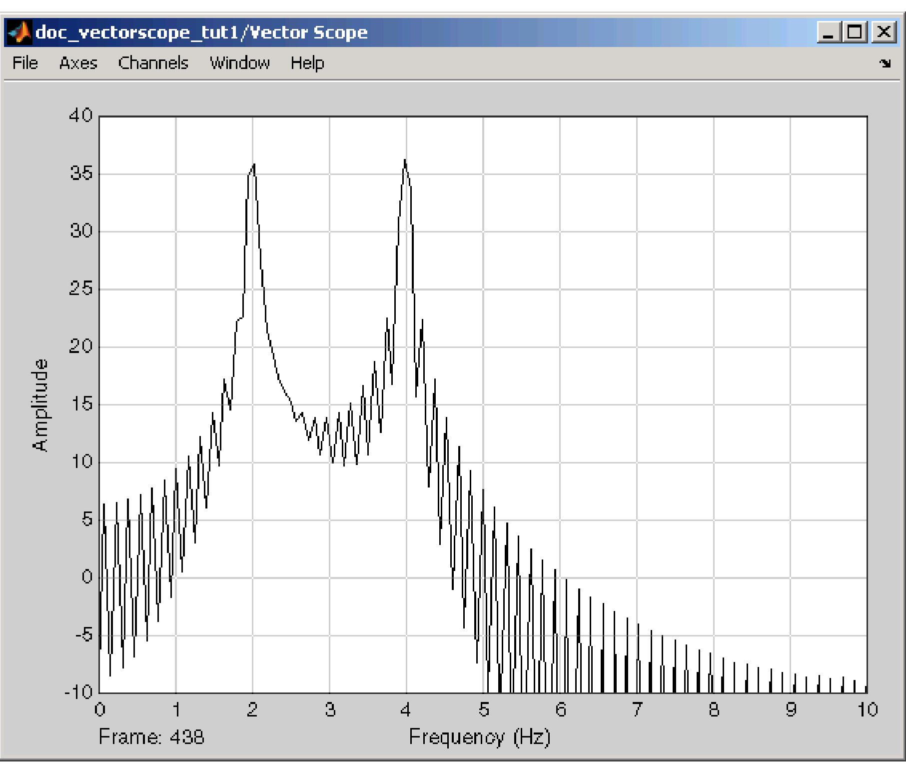
Based on these parameters, the Magnitude FFT block zero-pads the length-128 input frame to a length of 256 before performing the FFT.

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 Signal Processing Blockset 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.

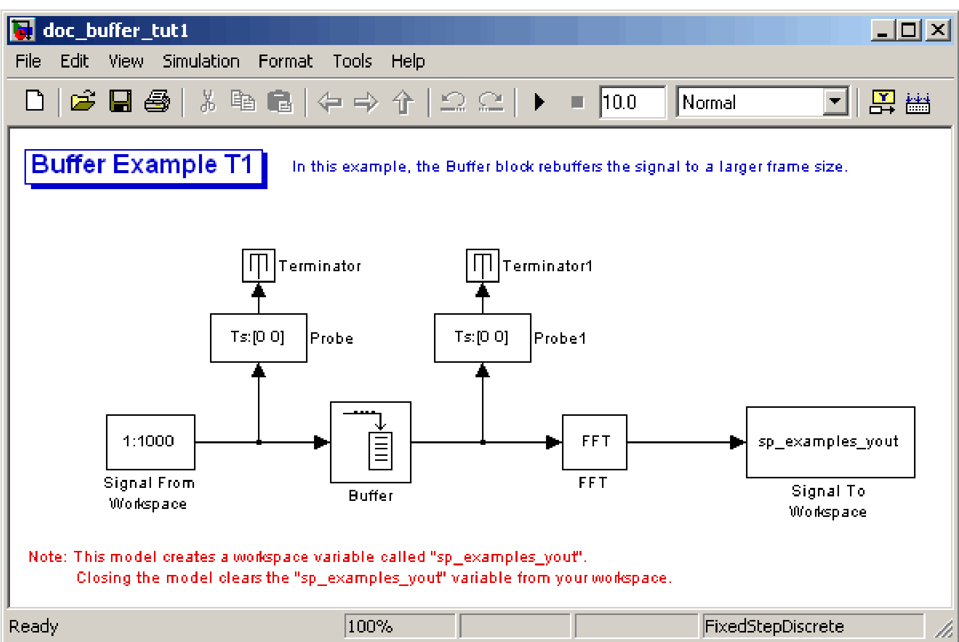
In all of these cases, the sample period of the output sequence is *not* equal to the sample period of the input sequence.

Buffering with Preservation of the Signal

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 `doc_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.

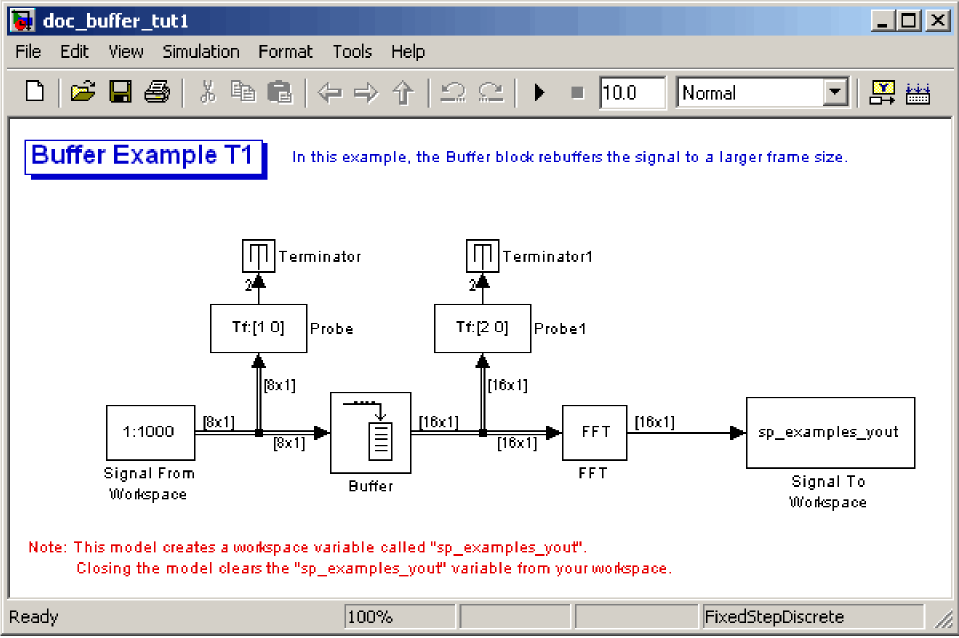
6 Set the parameters as follows, and then click **OK**:

- **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.



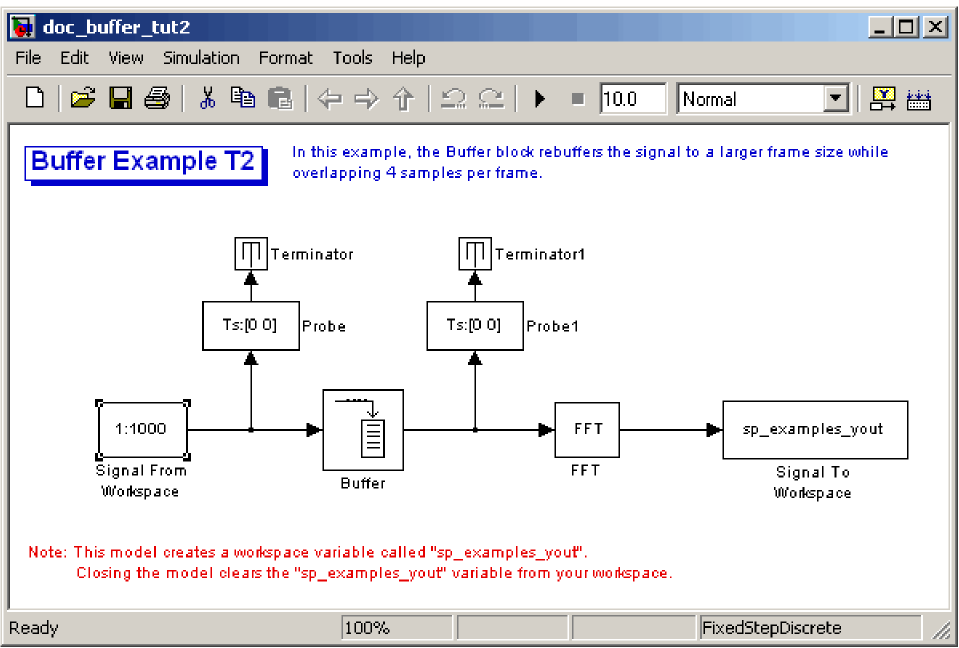
Note that the input to the Buffer block has a frame size of 8 and the output of the block has a frame size of 16. As shown by the Probe blocks, the rebuffering process doubles the frame period from 1 to 2 seconds.

Buffering with Alteration of the Signal

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 `doc_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.

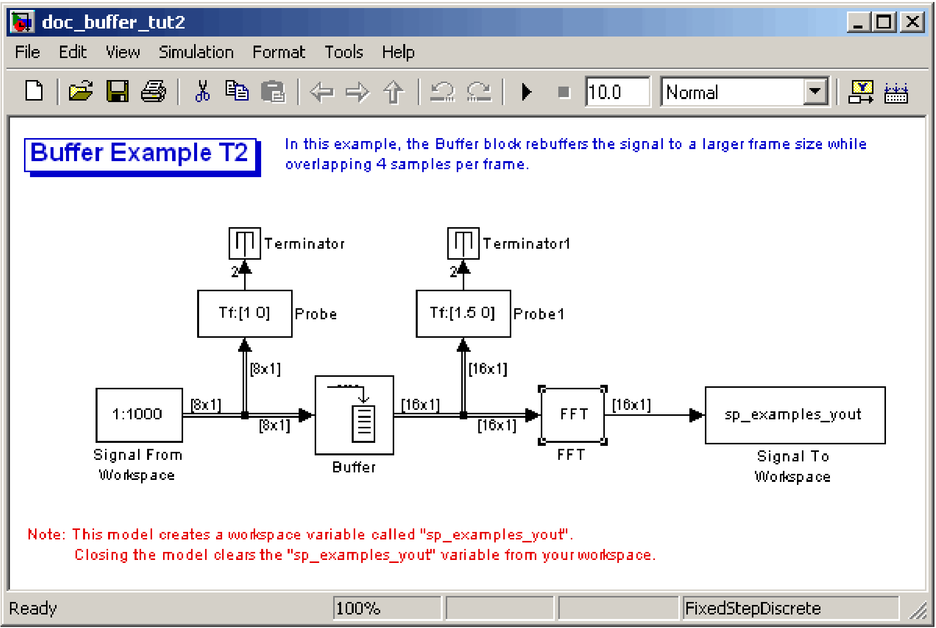
6 Set the parameters as follows, and then click **OK**:

- **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.

Converting Frame Status

In this section...

"Frame Status" on page 2-33

"Buffering Sample-Based Signals into Frame-Based Signals" on page 2-33

"Buffering Sample-Based Signals into Frame-Based Signals with Overlap" on page 2-37

"Buffering Frame-Based Signals into Other Frame-Based Signals" on page 2-41

"Buffering Delay and Initial Conditions" on page 2-44

"Unbuffering Frame-Based Signals into Sample-Based Signals" on page 2-45

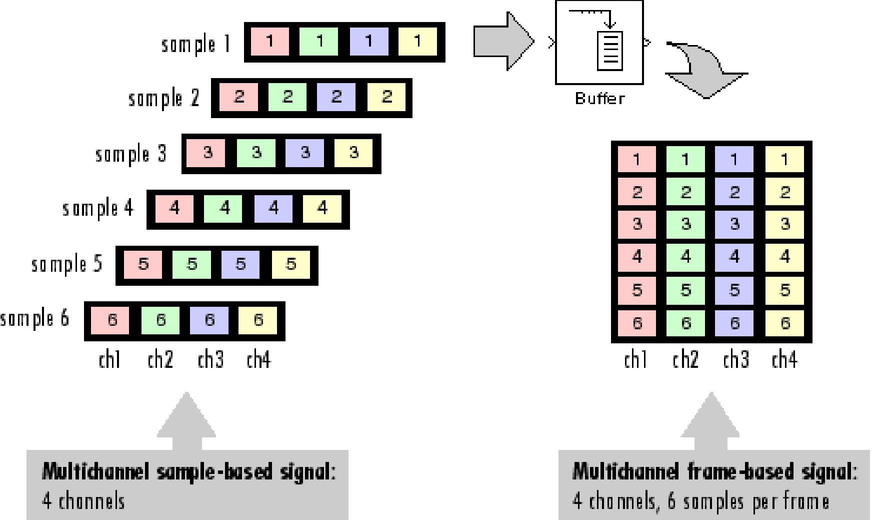
Frame Status

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 ,→, for a sample-based signal and a double line, ⇒ 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.

Buffering 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.

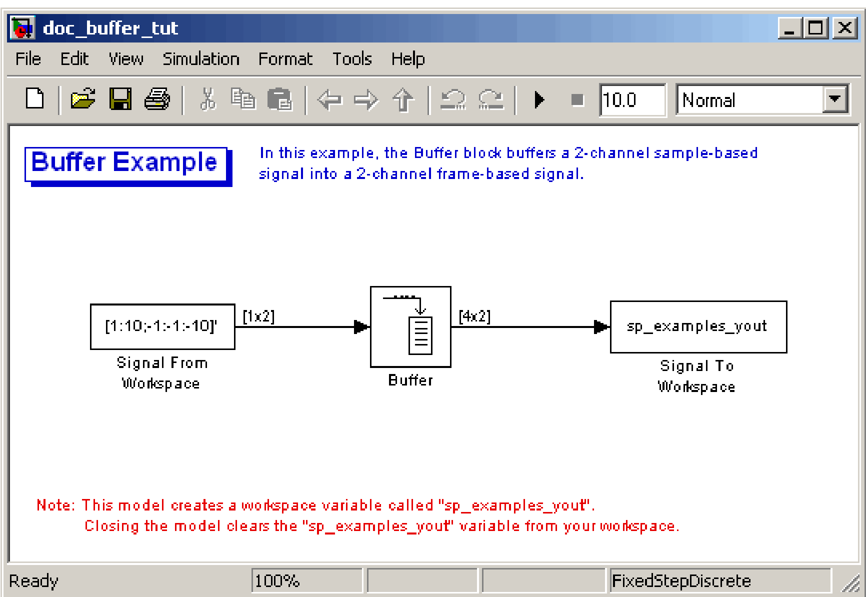
The following figure is a graphical representation of a sample-based signal being converted into a frame-based signal by the Buffer block.



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 `doc_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:

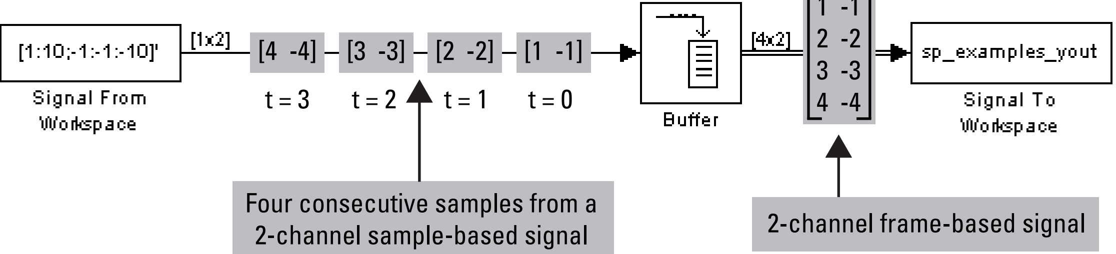
- **Output buffer size (per channel)** = 4
- **Buffer overlap** = 0
- **Initial conditions** = 0

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.

Buffering 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.

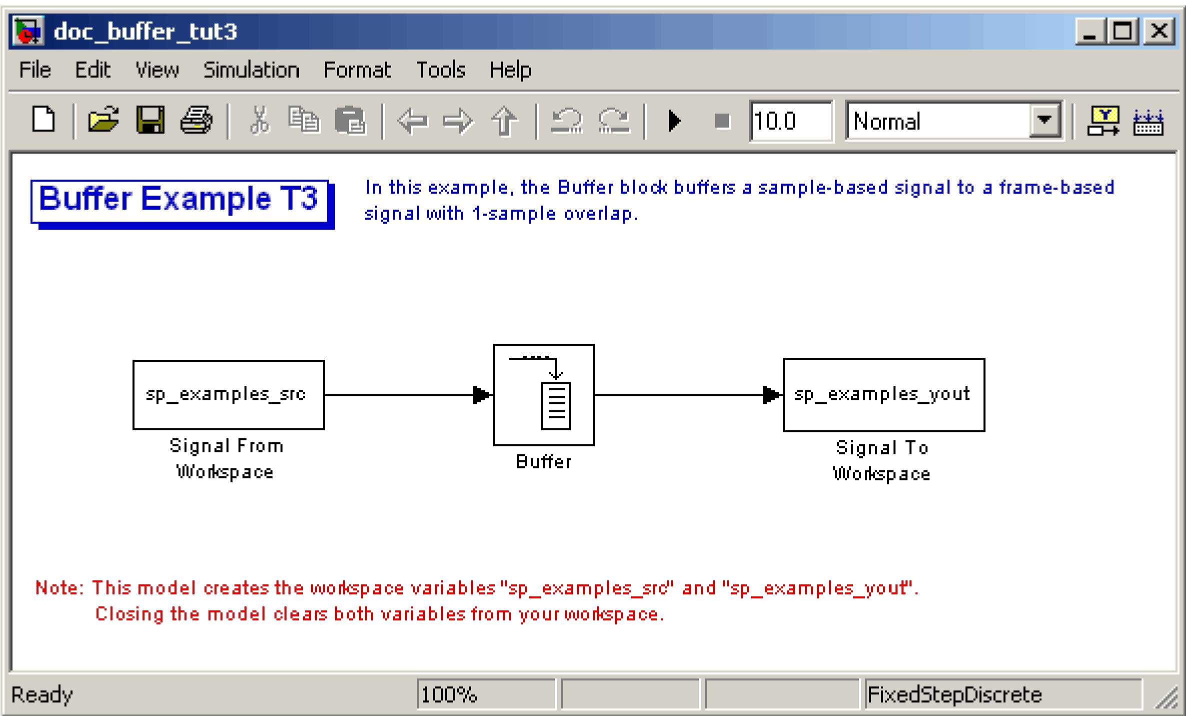
The **Buffer overlap** parameter of the Buffer block specifies the number of overlap points, L. In the overlap case ($L > 0$), the frame period for the output is $(M_o\text{-}L)*T_{si}$, where T_{si} is the input sample period and M_o is the **Buffer size**.

Note Set the **Buffer overlap** parameter to a negative value to achieve output frame rates *slower* than in the nonoverlapping case. The output frame period is still $T_{si}*(M_o\text{-}L)$, but now with $L < 0$. Only the M_o newest inputs are included in the output buffers. The previous L inputs are discarded.

In the following example, a four-channel 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 `doc_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**:

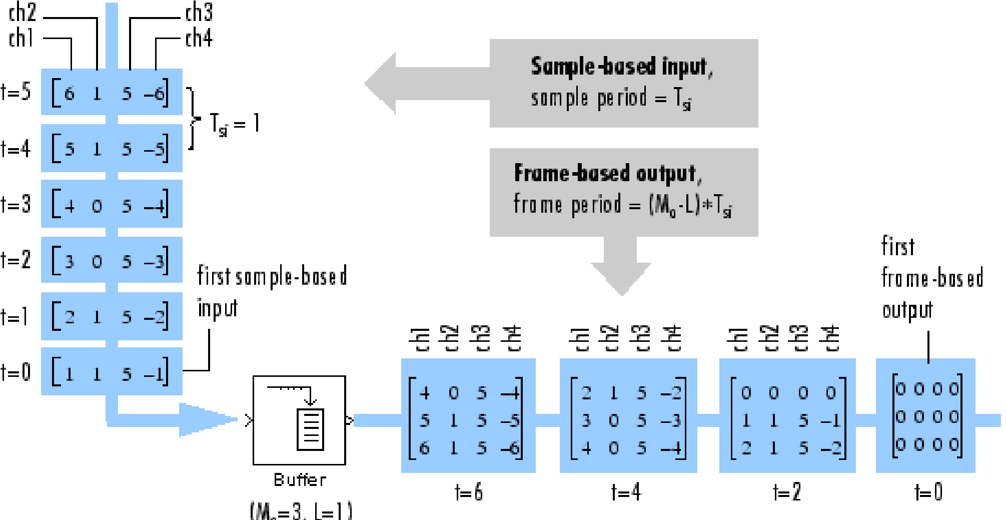
   - **Output buffer size (per channel)** = `3`
   - **Buffer overlap** = `1`
   - **Initial conditions** = `0`

   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.

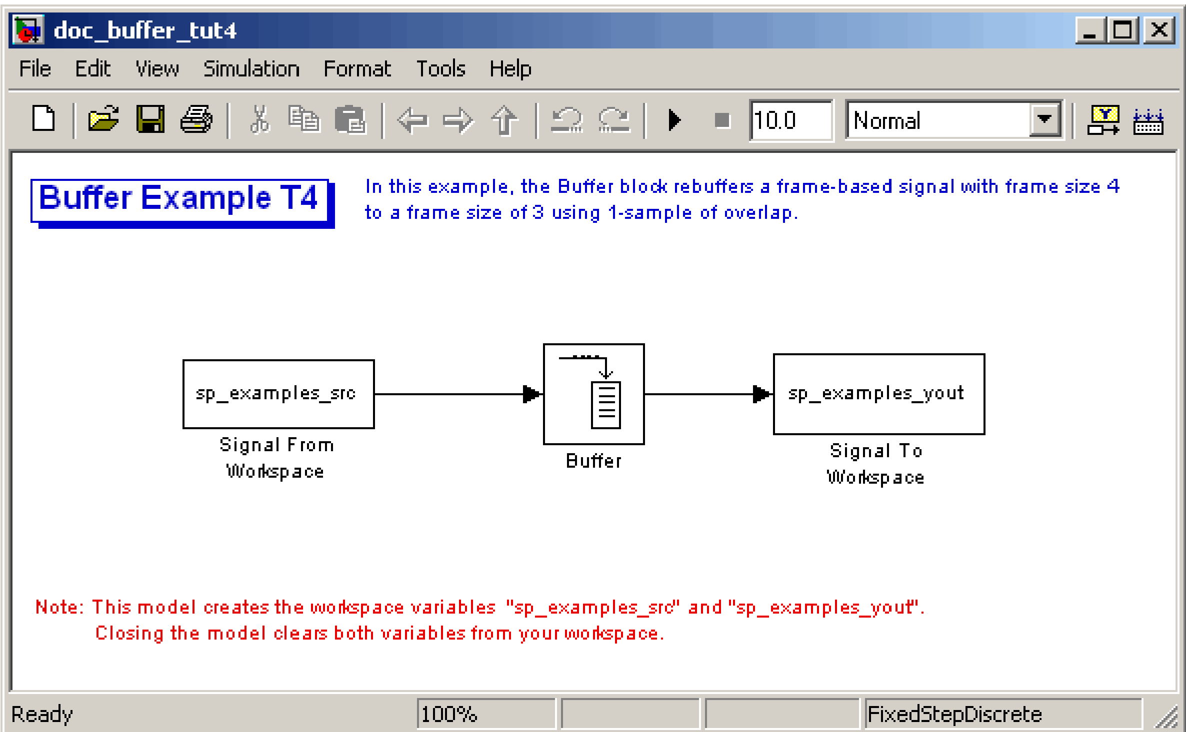
See "Excess Algorithmic Delay (Tasking Latency)" on page 2-57 for general information about algorithmic delay. For instructions on how to calculate buffering delay, see "Buffering Delay and Initial Conditions" on page 2-44.

Buffering 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 `doc_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.

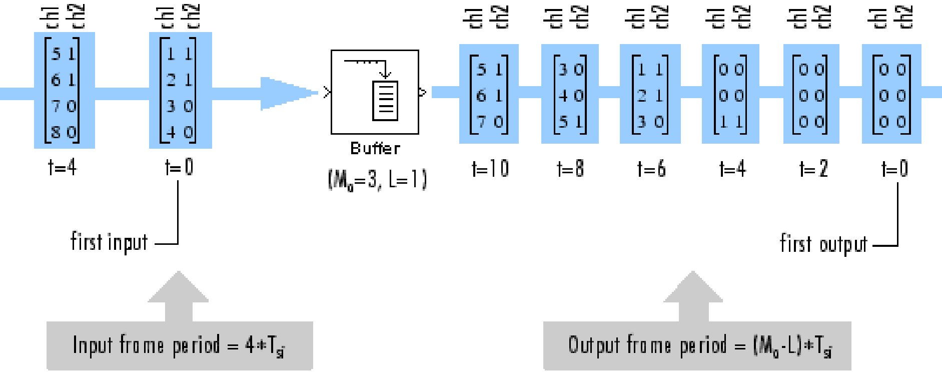
6 Set the block parameters as follows, and then click **OK**:

- **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 2-57 for general information about algorithmic delay. For instructions on how to calculate buffering delay, and see “Buffering Delay and Initial Conditions” on page 2-44.

Buffering Delay and Initial Conditions

In the examples “Buffering Sample-Based Signals into Frame-Based Signals with Overlap” on page 2-37 and “Buffering Frame-Based Signals into Other Frame-Based Signals” on page 2-41, 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 “Buffering Frame-Based Signals into Other Frame-Based Signals” on page 2-41 using the following command at the MATLAB command line:

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

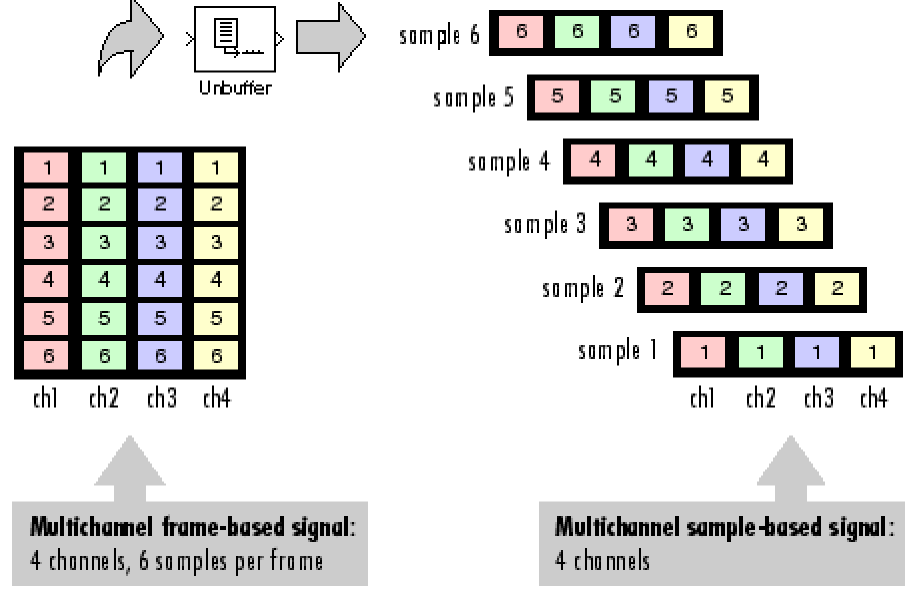
This result agrees with the block’s output in that example. Notice that this model was simulated in Simulink multitasking mode.

For more information about delay, see “Excess Algorithmic Delay (Tasking Latency)” on page 2-57. 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`.

Unbuffering 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.

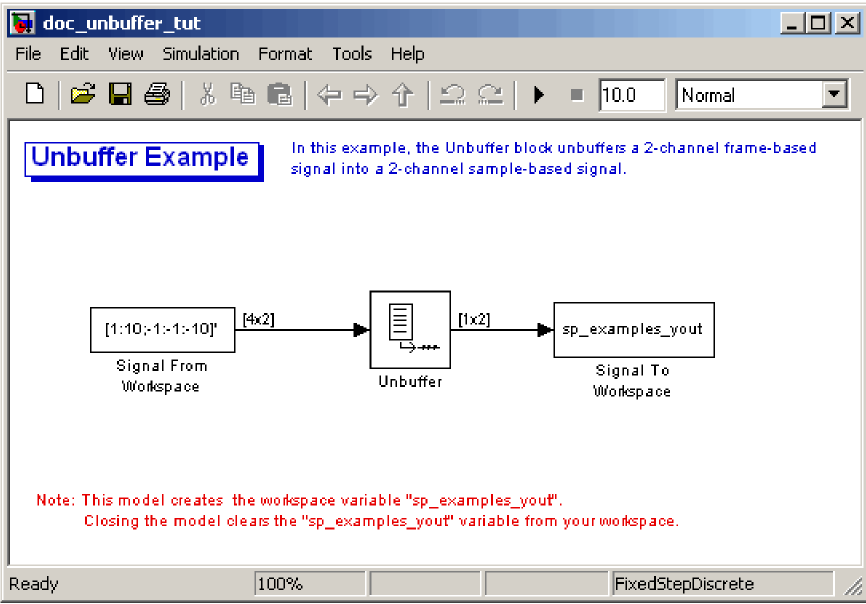
$$T_{so} = T_{fi} / M_i$$

The Unbuffer block always preserves the signal's sample period ($T_{so} = T_{si}$). See "Converting Sample and Frame Rates" on page 2-11 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 `doc_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**.

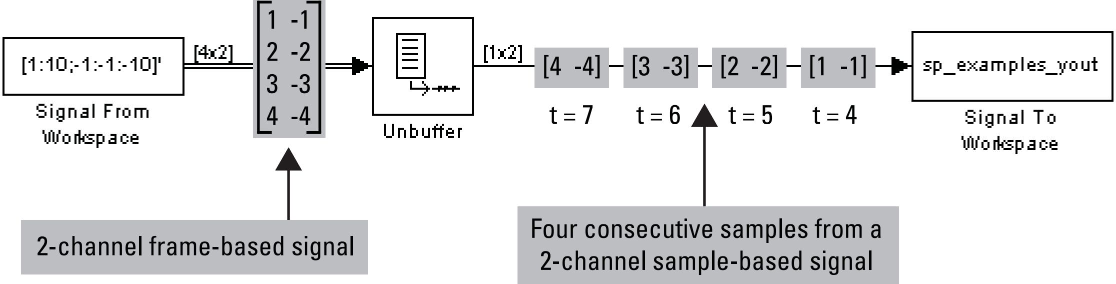
5 Double-click the Unbuffer block. The **Block Parameters: Unbuffer** dialog box opens.

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 figures is a graphical representation of what happens during the model simulation.



Note The Unbuffer block generates initial conditions not shown in the figure below with the value specified by the **Initial conditions** parameter. See the Unbuffer reference page for information about the number of initial conditions that appear in the output.

---

8 At the MATLAB command prompt, type `sp_examples_yout`.

The following is a portion of the output.

```
sp_examples_yout(:,:,1) =

        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

In this section...

"Computational Delay" on page 2-49
"Algorithmic Delay" on page 2-51
"Zero Algorithmic Delay" on page 2-51
"Basic Algorithmic Delay" on page 2-54
"Excess Algorithmic Delay (Tasking Latency)" on page 2-57
"Predicting Tasking Latency" on page 2-59

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.

Reducing 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 1-16 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 *Signal Processing Blockset 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 Real-Time Workshop 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 Real-Time Workshop 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 *Signal Processing Blockset Reference* and elsewhere 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 2-51
- "Basic Algorithmic Delay" on page 2-54
- "Excess Algorithmic Delay (Tasking Latency)" on page 2-57

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

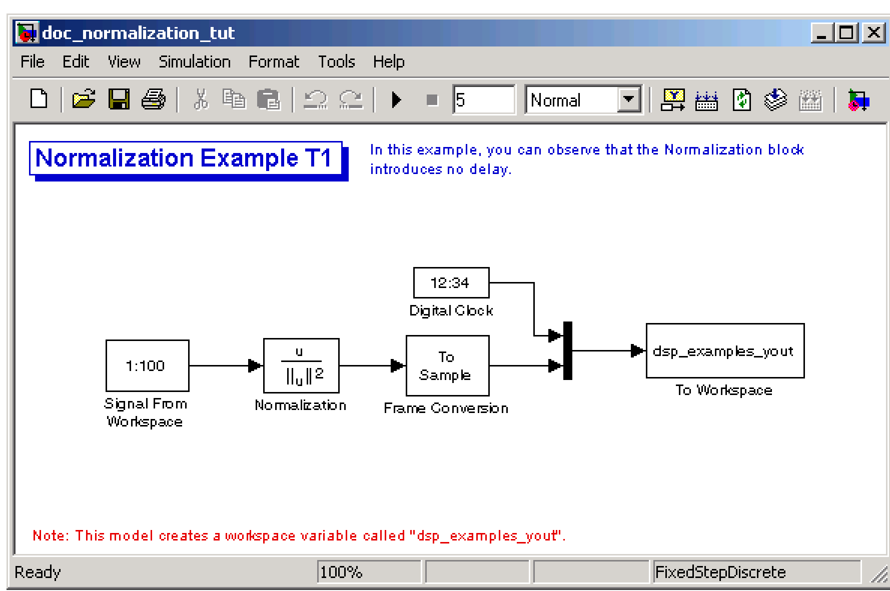
The FFT block is an example of a component that has no algorithmic delay. The Simulink timer does not record any passage of time while the block computes the FFT of the input, and the transformed data is available at the output in the same time step that the input is received. There are many other blocks 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.

In the *Signal Processing Blockset Reference* blocks are assumed to have zero delay unless otherwise indicated. If a block has zero delay for one combination of parameter settings but nonzero delay for another, the block reference page contains this fact.

The Normalization block is an example of a block with zero algorithmic delay:

1. At the MATLAB command prompt, type `doc_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

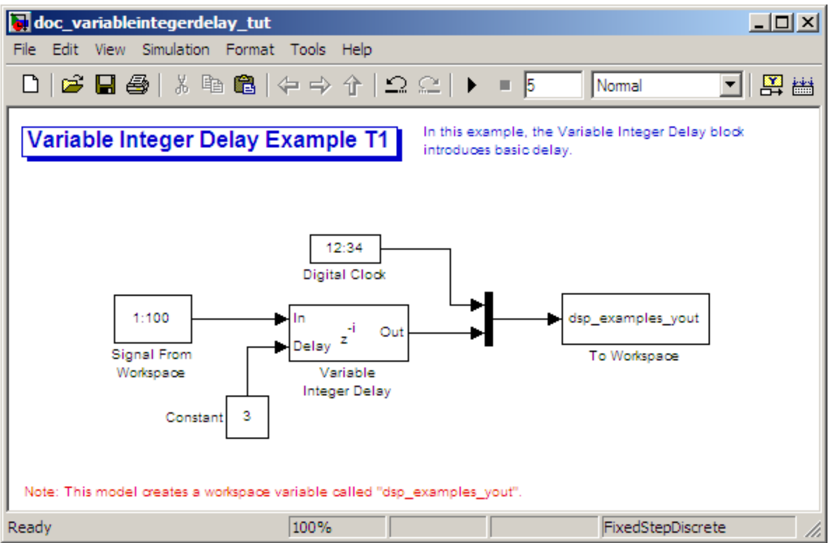
When several blocks with zero algorithmic delay are connected in a feedback loop, Simulink may report an algebraic loop error and performance may generally suffer. You can prevent algebraic loops by injecting at least one sample of delay into a feedback loop , for example, by including a Delay block 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 `doc_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.

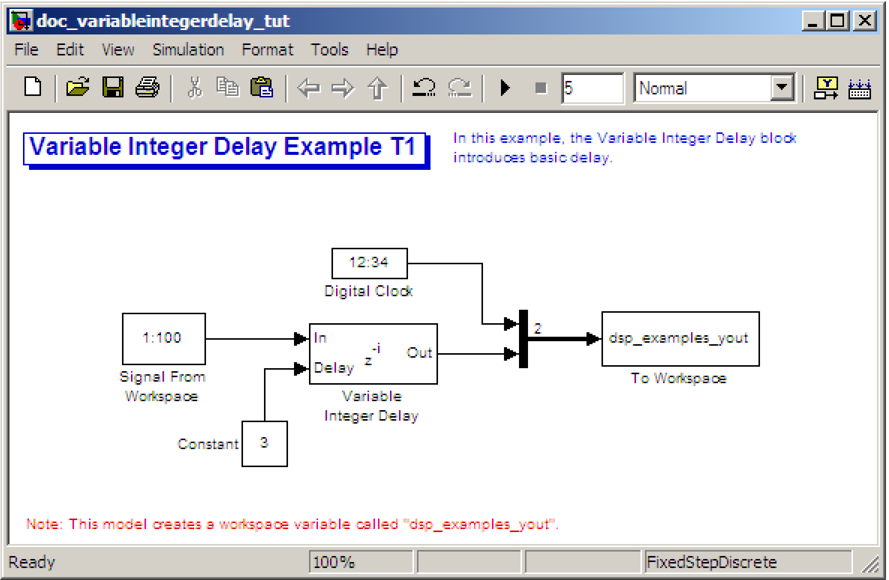
7 Double-click the Variable Integer Delay block. The **Block Parameters: Variable Integer Delay** dialog box opens.

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 Signal Processing Blockset 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 Signal Processing Blockset blocks.

Excess Algorithmic Delay (Tasking Latency)

Under certain conditions, Simulink may force a block to delay inputs longer than is strictly required by the block's algorithm. This excess algorithmic delay is called tasking latency, because it arises from synchronization requirements of the Simulink tasking mode. A block's overall algorithmic delay is the sum of its basic delay and tasking latency.

Algorithmic delay = Basic algorithmic delay + Tasking latency

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

- "Simulink Tasking Mode" on page 2-57
- "Block Rate Type" on page 2-58
- "Model Rate Type" on page 2-58
- "Block Sample Mode" on page 2-59

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 Considerations" in the *Real-Time Workshop 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

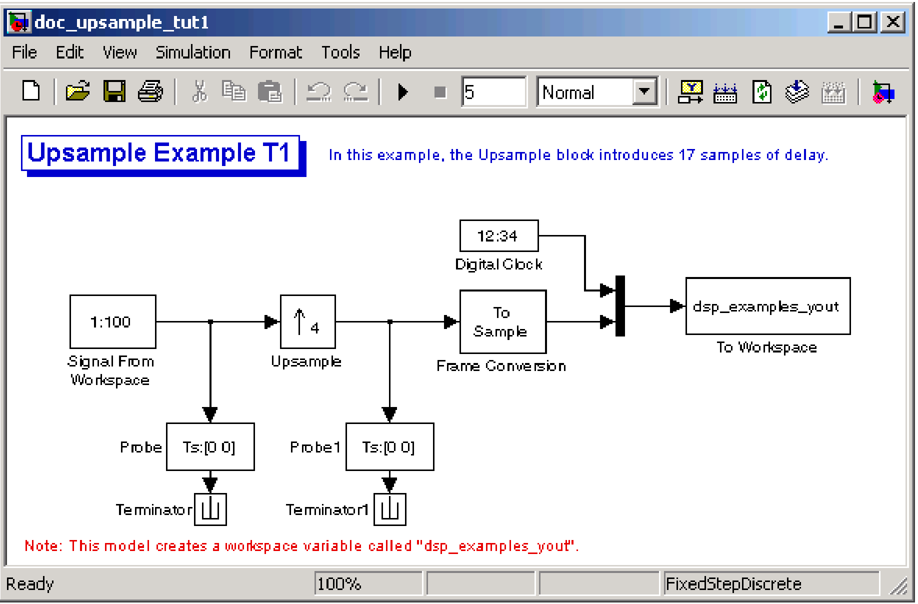
Many blocks can operate in either sample-based or frame-based modes. In source blocks, the mode is usually determined by the **Samples per frame** parameter. If, for the **Samples per frame** parameter, you enter 1, the block operates in sample-based mode. If you enter a value greater than 1, the block 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.

Predicting 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 `doc_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`
   - **Frame-based mode** = `Maintain input frame size`

   The **Frame-based mode** 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**.

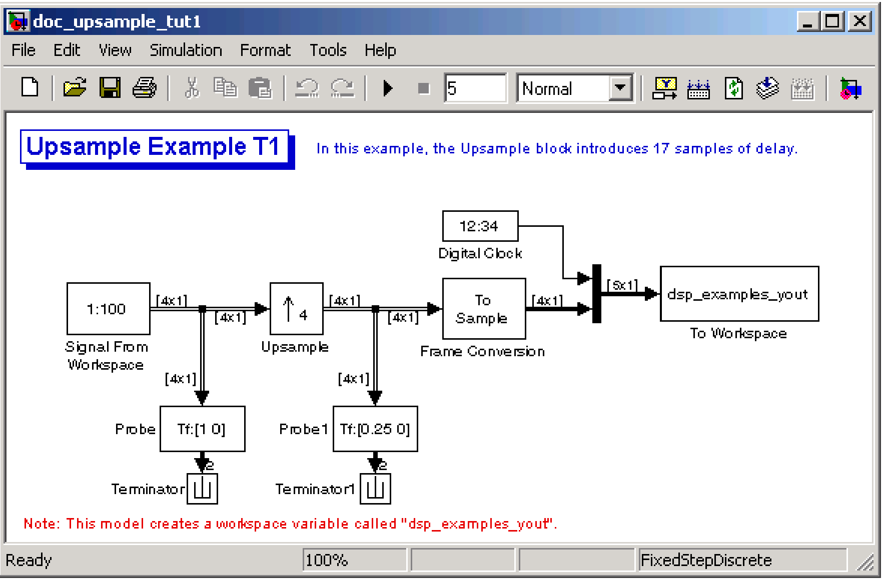
    This matches the sample period of the Upsample block's output.

11 Double-click the Frame Conversion block. The **Block Parameters: Frame Conversion** dialog box opens.

12 Set the **Output signal** parameter of the 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 =
```

time					
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	

"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.

Filters

The Signal Processing Blockset Filtering library provides an extensive array of filtering blocks for designing and implementing filters in your models.

- "Digital Filter Block" on page 3-2
- "Digital Filter Design Block" on page 3-17
- "Filter Realization Wizard" on page 3-31
- "Analog Filter Design Block" on page 3-51
- "Adaptive Filters" on page 3-53
- "Multirate Filters" on page 3-66

Digital Filter Block

In this section...

"Overview of the Digital Filter Block" on page 3-2

"Implementing a Lowpass Filter" on page 3-3

"Implementing a Highpass Filter" on page 3-4

"Filtering High-Frequency Noise" on page 3-5

"Specifying Static Filters" on page 3-9

"Specifying Time-Varying Filters" on page 3-10

"Specifying the SOS Matrix (Biquadratic Filter Coefficients)" on page 3-15

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 "Real-Time Workshop" 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 predesigned filter. Both blocks implement a filter in the same manner and have the same behavior during simulation and code generation.

Implementing a Lowpass Filter

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

   ```
   lopassNum = [-0.0021 -0.0108 -0.0274 -0.0409 -0.0266 0.0374
   0.1435 0.2465 0.2896 0.2465 0.1435 0.0374 -0.0266 -0.0409
   -0.0274 -0.0108 -0.0021];
   ```

2. Open Simulink and create a new model file.

3. From the Signal Processing Blockset 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** = `lopassNum`
   - **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 `lopassNum`.
   - Type in filter design commands from Signal Processing Toolbox™ software or Filter Design 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, "Implementing a Highpass Filter" on page 3-4, 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 3-17.

Implementing a Highpass Filter

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

1. If the model you created in "Implementing a Lowpass Filter" on page 3-3 is not open on your desktop, you can open an equivalent model by typing

   ```
   doc_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 Signal Processing Blockset 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`
   - **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 Filter Design 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, "Filtering High-Frequency Noise" on page 3-5, 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 3-17.

Filtering High-Frequency Noise

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 "Implementing a Highpass Filter" on page 3-4 is not open on your desktop, you can open an equivalent model by typing

```
doc_filter_ex2
```

at the MATLAB command prompt.

2 If you have not already done so, define the lowpass and highpass filter coefficients in the MATLAB workspace by typing

```
lopassNum = [-0.0021 -0.0108 -0.0274 -0.0409 -0.0266 ...
0.0374 0.1435 0.2465 0.2896 0.2465 0.1435 0.0374 ...
-0.0266 -0.0409 -0.0274 -0.0108 -0.0021];
hipassNum = [-0.0051 0.0181 -0.0069 -0.0283 -0.0061 ...
0.0549 0.0579 -0.0826 -0.2992 0.5946 -0.2992 -0.0826 ...
0.0579 0.0549 -0.0061 -0.0283 -0.0069 0.0181 -0.0051];
```

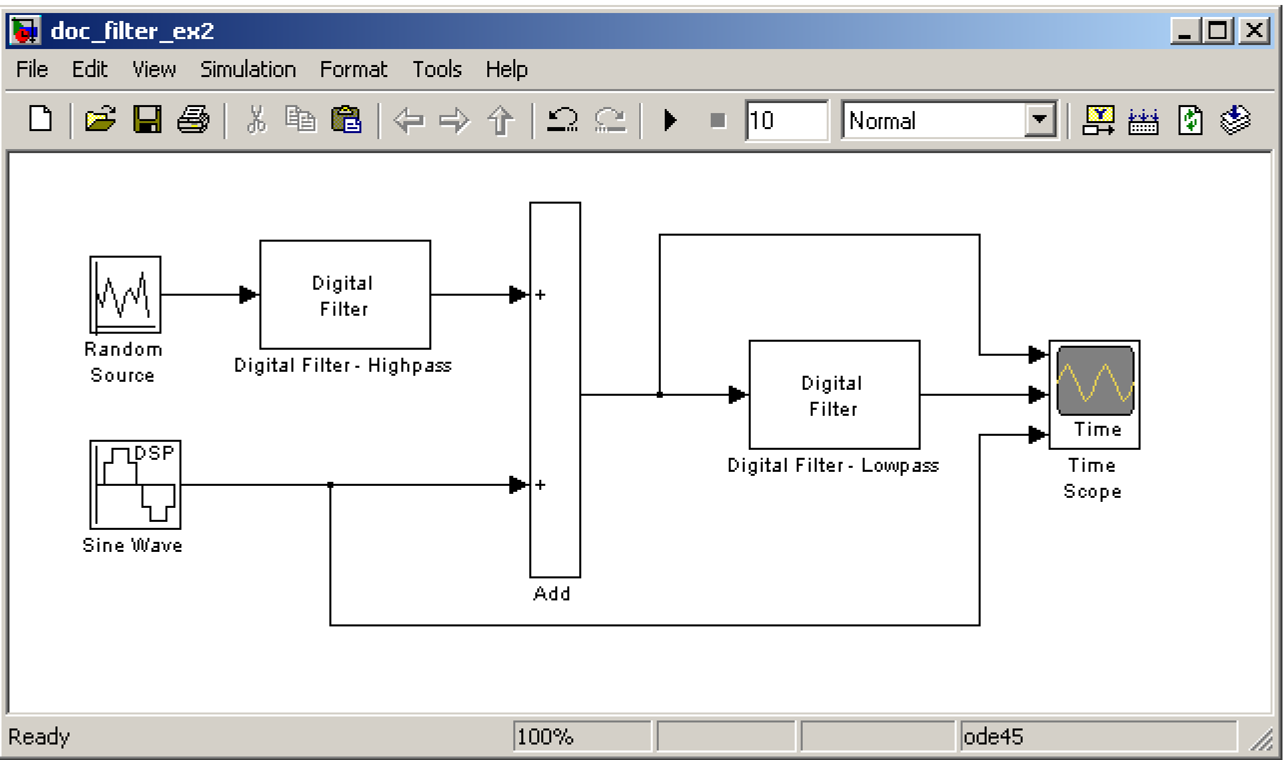
3 Click-and-drag the following blocks into your model file.

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

4 Set the parameters for the rest of the blocks as indicated in the following table. For any parameters not listed in the table, leave them at their default settings.

Block	Parameter Setting
Add	• **Icon shape** = `rectangular` • **List of signs** = `++`
Random Source	• **Source type** = `Uniform` • **Minimum** = `0` • **Maximum** = `4` • **Sample mode** = `Discrete` • **Sample time** = `1/1000` • **Samples per frame** = `50`
Sine Wave	• **Frequency (Hz)** = `75` • **Sample time** = `1/1000` • **Samples per frame** = `50`
Time Scope	• **File > Number of Input Ports > 3** • **File > Configuration ...** – Open the **Visuals:Time Domain Options** dialog and set **Time range** = `Input sample time`

5 Connect the blocks as shown in the following figure. You 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:

- Channel 1 = `Noisy Sine Wave`
- Channel 2 = `Filtered Noisy Sine Wave`
- Channel 3 = `Original Sine Wave`

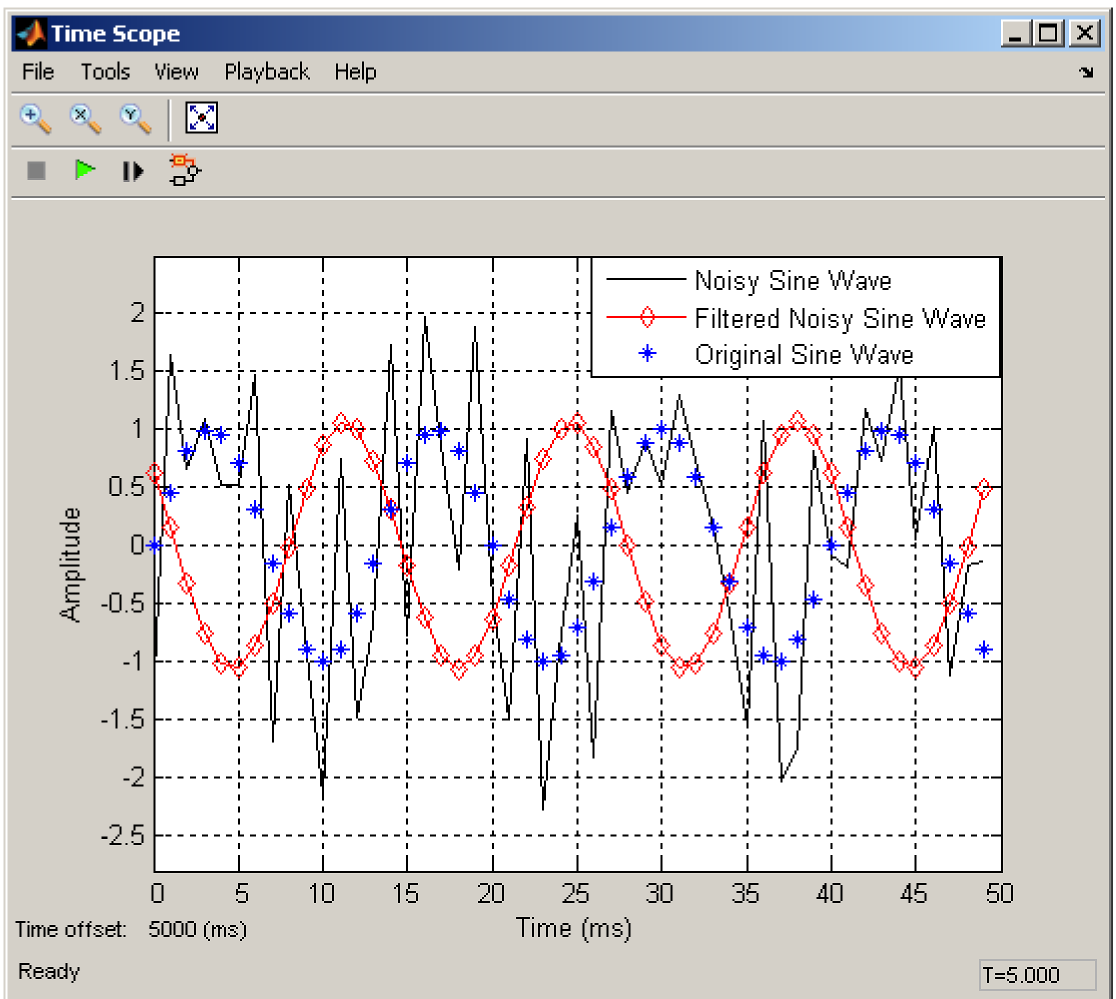
In the next step, you will set the color, style, and marker of each channel.

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 3-17.

Specifying 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 *Signal Processing Blockset 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.

Specifying 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. To learn more, see "Setting the Coefficient Update Rate" on page 3-11.

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*. To learn more, see "Providing Filter Coefficient Vectors at Block Input Ports" on page 3-12.

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 3-14.

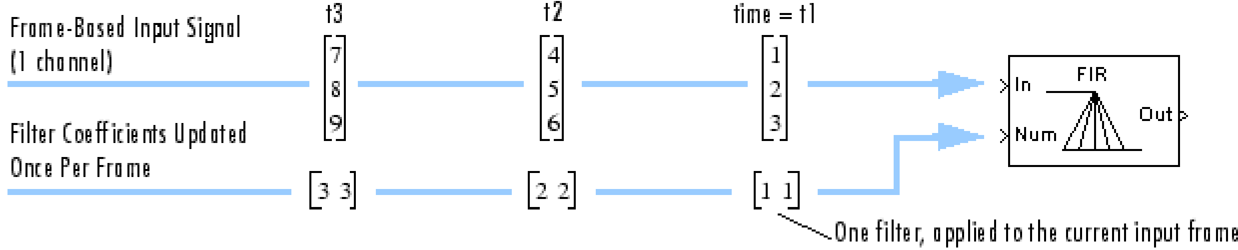
Setting the Coefficient Update Rate

When the input is frame based, the block updates time-varying filters once every input frame, or once for every sample in an input frame, depending on the **Coefficient update rate** parameter:

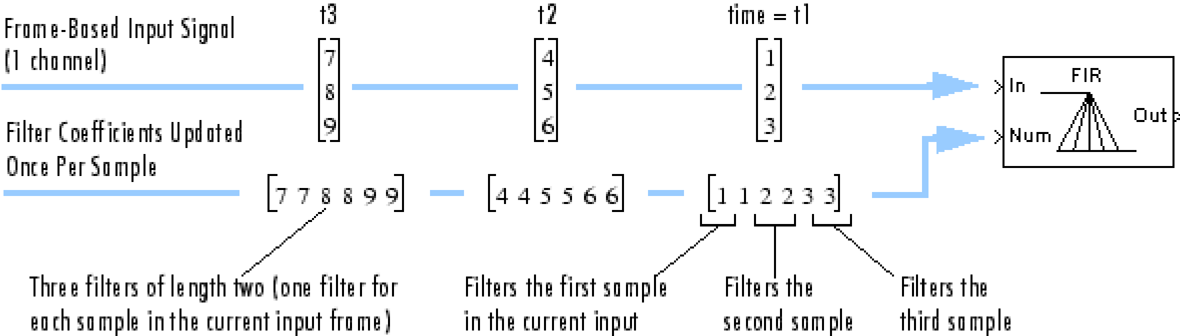
- `One filter per frame` — Each coefficient vector represents one filter that is applied to all samples in the current frame.
- `One filter per sample` — Each coefficient vector represents a concatenation of filter coefficients. When you have N samples per frame and M coefficients for each filter, then the coefficient vector length is M*N. All the coefficient vectors must be of equal length.

The following figure shows the block filtering one channel; however, the block can filter multiple channels. Note that the block *can* apply a single filter to multiple channels, but *cannot* apply a different filter to each channel.

Update filter coefficients once per frame:
At time t1, the block applies the filter [1 1] to all three samples in the first frame of input data.
At time t2, the block updates the filter to [2 2] and applies it to the second frame of data, and so on.



Update filter coefficients once per sample:
At time t1, the block applies filter [1 1] to the first sample in the first frame of data, filter [2 2] to the second sample, and filter [3 3] to the third sample. At time t2, the block updates the filter for each sample in the next input frame, and applies each filter to the corresponding sample, and so on. The block preserves state from sample to sample.



Providing Filter Coefficient Vectors at Block Input Ports

As illustrated in the previous figure, the filter coefficient vectors for filters that update once per frame are different from coefficient vectors for filters that update once per sample. See the following tables to meet the rate and length requirements of the filter coefficient vectors:

- Length requirements — See the table Length Requirements for Time-Varying Filter Coefficient Vectors on page 3-13.
- Rate requirements — See the table Rate Requirements for Time-Varying Filter Coefficient Vectors on page 3-14.

The output size, frame status, and dimension always match those of the input signal that is filtered, not the vector of filter coefficients.

Length Requirements for Time-Varying Filter Coefficient Vectors

Coefficient Update Rate	How to Specify Filter Coefficient Vectors (Also see the previous figure)	Length Requirements
Once per frame	Each coefficient vector corresponds to one input frame and represents one filter. Specify each vector as you would any static filter: $[b_0, b_1, b_2, ..., b_n]$, $[a_0, a_1, a_2, ..., a_m]$, or $[k_1, k_2, ..., k_n]$	None
Once per sample	Each coefficient vector corresponds to one input frame. However, the vector represents multiple filters of the same length with one filter for each sample in the current frame. To create such a vector, concatenate all the filters for each sample within the input frame. For instance, the following vector specifies length-2 numerator coefficients for each sample in a three-sample frame $[b_0\ b_1\ B_0\ B_1\ \beta_0\ \beta_1]$ where $[b_0\ b_1]$ filters the first sample in the input frame, $[B_0\ B_1]$ filters the second sample, and so on.	All filters must be the same length, L. The length of each filter coefficient vector must be L times the number of samples per frame in the input. (Each sample in the frame has one set of filter coefficients.)

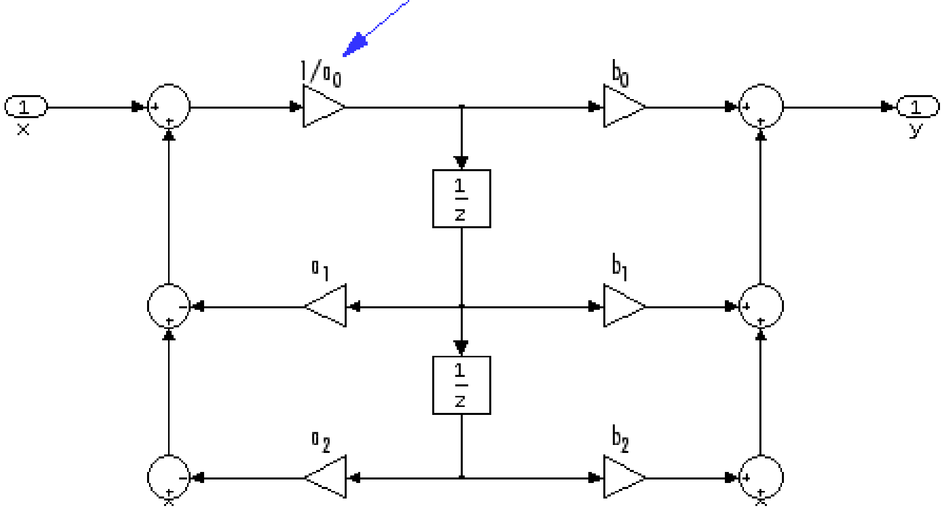
The time-varying filter coefficient vectors can be sample- or frame-based row or column vectors. The vectors of filter coefficients must arrive at their input port at the same times that the frames of input data arrive at their input port, as indicated in the following table.

Rate Requirements for Time-Varying Filter Coefficient Vectors

Input Signal	Time-Varying Filter Coefficient Vectors	Rate Requirements (Also see the previous figure)
Sample based	Sample based	Sample rates of input and filter coefficients must be equal.
Sample based	Frame based	Input sample rate must equal filter coefficient frame rate.
Frame based	Sample based	Input frame rate must equal filter coefficient sample rate.
Frame based	Frame based	Frame rates of input and filter coefficients must be equal.

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.



Specifying 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:

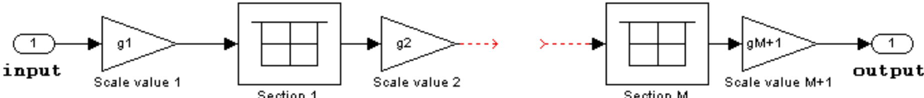
- **Transfer function type** — `IIR (poles & zeros)`
- **Filter structure** — `Biquad direct form I (SOS)`, or `Biquad direct form I transposed (SOS)`, or , or `Biquad direct form II transposed (SOS)`
- **SOS matrix (Mx6)** M-by-6 SOS matrix

  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.

Digital Filter Design Block

In this section...

"Overview of the Digital Filter Design Block" on page 3-17
"Choosing Between Filter Design Blocks" on page 3-18
"Creating a Lowpass Filter" on page 3-21
"Creating a Highpass Filter" on page 3-23
"Filtering High-Frequency Noise" on page 3-25

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 "Real-Time Workshop" to generate C code from your filter block. For more information on generating C code from models, see "Code Generation" in the *Signal Processing Blockset Getting Started Guide*.

Alternatively, you can use other MathWorks™ products, such as Signal Processing Toolbox software and Filter Design Toolbox software, to design your filters. Once you design a filter using either toolbox, you can use one of the filter implementation blocks from Signal Processing Blockset software, such as the Digital Filter block, to realize the filters in your models. For more information, see the Signal Processing Toolbox documentation and Filter Design Toolbox documentation. To learn how to import and export your filter designs, see "Importing and Exporting Quantized Filters" in the Filter Design Toolbox documentation.

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 specified. 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 Filter Design 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 "Importing and Exporting Quantized Filters" in the Filter Design Toolbox documentation.

Note Use the Digital Filter Design block to design and implement a filter. Use the Digital Filter block to implement a predesigned filter. 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 "Choosing Between Filter Design Blocks" on page 3-18.

Choosing Between Filter Design Blocks

You can design and implement digital filters using the Digital Filter Design block and Filter Realization Wizard. This topic explains the similarities and differences between these blocks. In addition, you learn how to choose the block that is best suited for your needs.

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:

- Filter implementation method
  - The Digital Filter Design block opens the FDATool GUI to the **Design Filter** panel. It implements filters using the Digital Filter block. These filters are optimized for both speed and memory use in simulation and in C code generation. For more information on code generation, see "Code Generation" in the *Signal Processing Blockset Getting Started Guide*.
  - The Filter Realization Wizard opens the FDATool GUI to the **Realize Model** panel. The block can implement filters in two different ways. It can use the Simulink Sum, Gain, and Delay blocks, or it can use the Digital Filter block. If you choose to implement your filter using the Digital Filter block, your filter is bound by the type of filters this block supports.

Note If your filter is implemented by the Filter Realization Wizard using Sum, Gain, and Delay blocks, inputs to the filter must be sample based.

- 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.

- Multichannel filtering — The Digital Filter Design block can filter multichannel signals. Filters implemented by the Filter Realization Wizard can only filter single-channel signals.
- Data type support — The Digital Filter 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.

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 filter multichannel signals.
  - Use to generate highly optimized ANSI® C code that implements floating-point filters for embedded systems. For more information on code generation, see “Code Generation” in the *Signal Processing Blockset Getting Started Guide*.
- 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 generate multiple filter blocks rapidly.

See “Filter Realization Wizard” on page 3-31 and the Filter Realization Wizard block reference page for information.

Creating a Lowpass Filter

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 Signal Processing Blockset Filtering library, and then from the Filter Implementations library, click-and-drag a Digital Filter Design block into your model.

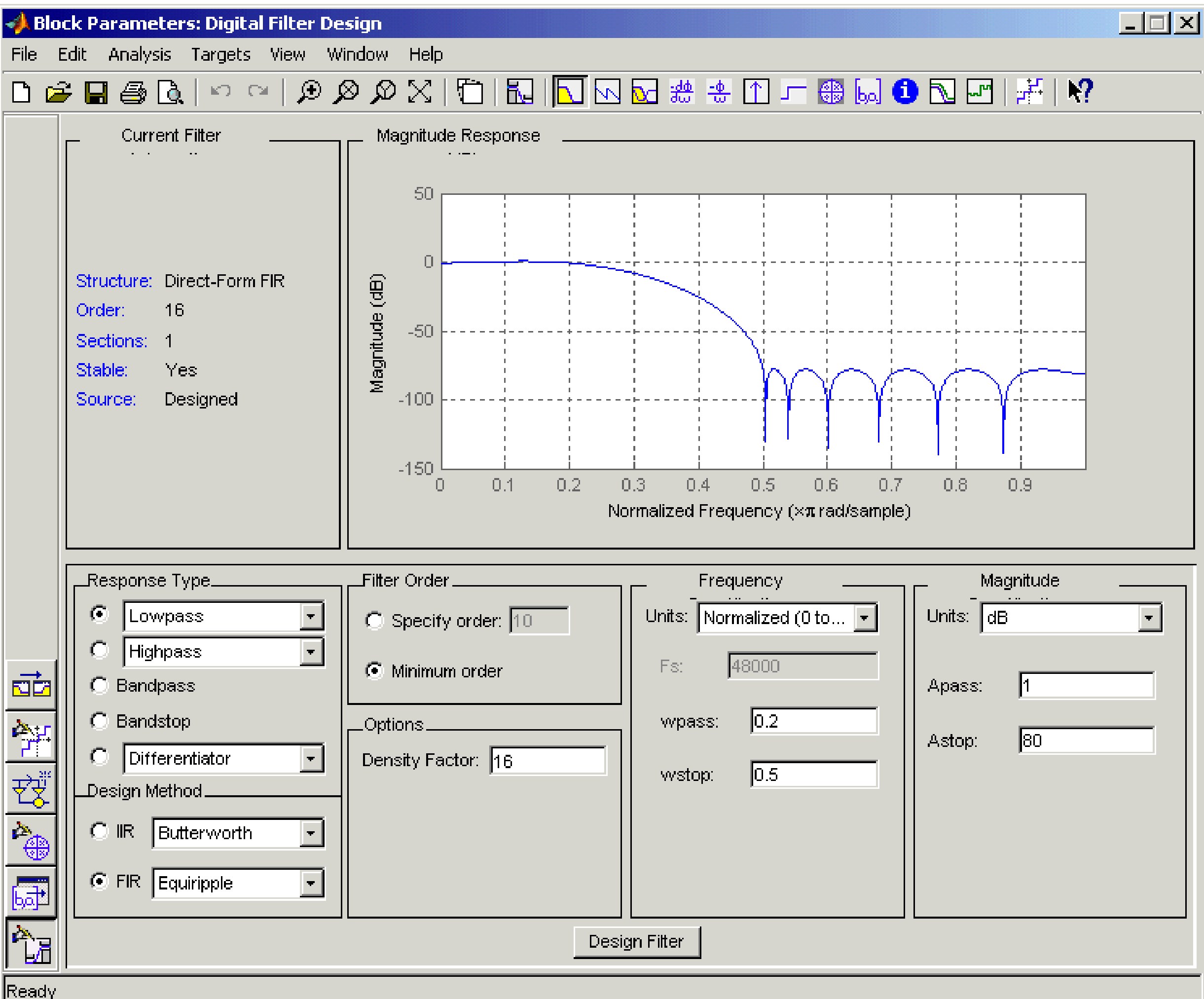
3. Double-click the Digital Filter Design block.

   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`

When you are finished, the GUI should look similar to the following figure:



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, "Creating a Highpass Filter" on page 3-23, you use a Digital Filter Design block to create a highpass filter. For more information about implementing a predesigned filter, see "Digital Filter Block" on page 3-2.

Creating a Highpass Filter

In this topic, you create a highpass filter using the Digital Filter Design block:

1 If the model you created in "Creating a Lowpass Filter" on page 3-21 is not open on your desktop, you can open an equivalent model by typing

```
doc_filter_ex4
```

   at the MATLAB command prompt.

2 From the Signal Processing Blockset 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.

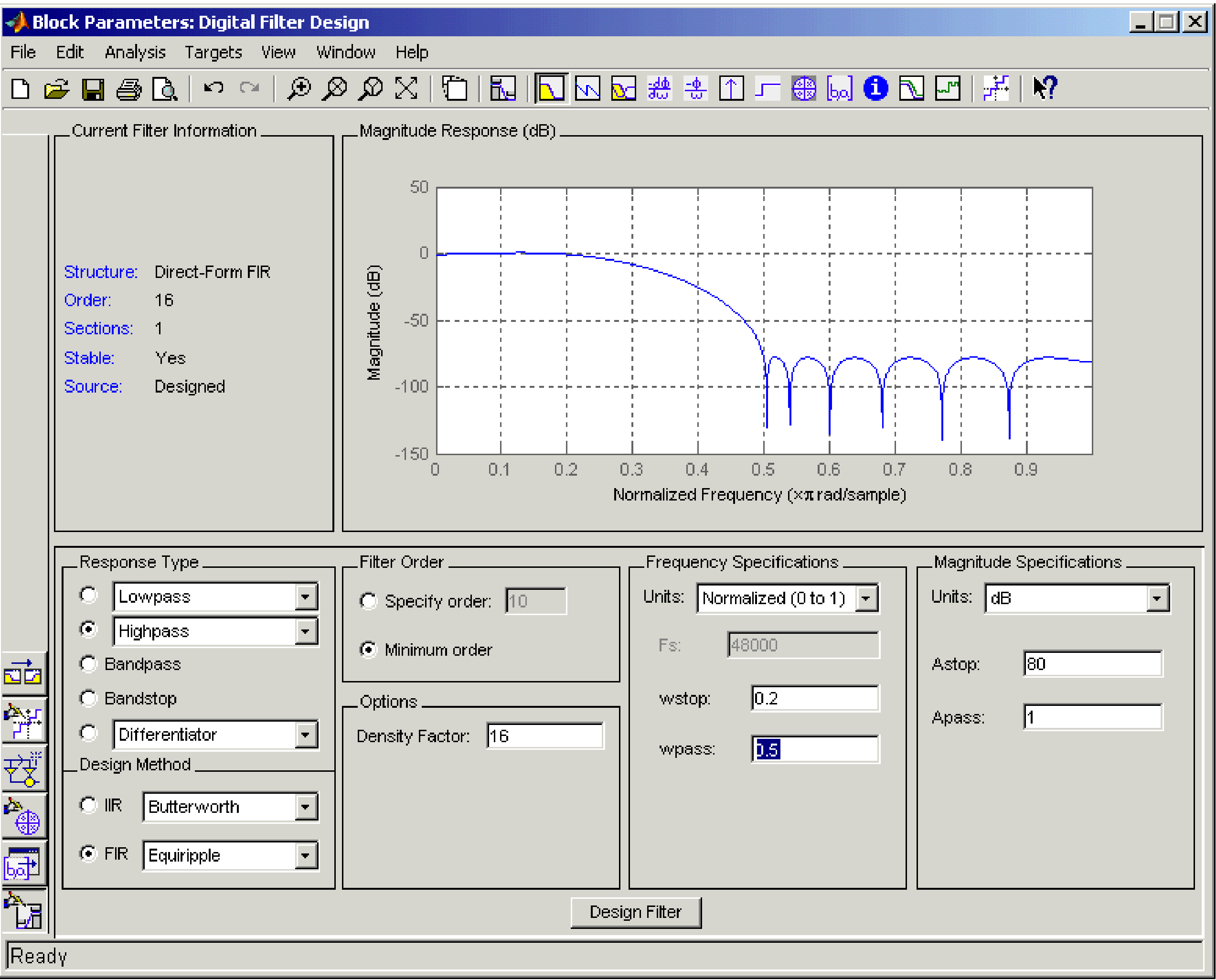
   The Filter Design and Analysis Tool (FDATool) GUI opens.

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`

When you are finished, the GUI should look similar to the following figure.



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 "Creating a Lowpass Filter" on page 3-21. The highpass filter passes the frequencies stopped by the lowpass filter, and stops the frequencies passed by the lowpass filter. In the next topic, "Filtering High-Frequency Noise" on page 3-25, 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 predesigned filter, see "Digital Filter Block" on page 3-2.

Filtering High-Frequency Noise

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 "Creating a Highpass Filter" on page 3-23 is not open on your desktop, you can open an equivalent model by typing

   ```
   doc_filter_ex5
   ```

   at the MATLAB command prompt.

2 Click-and-drag the following blocks into your model.

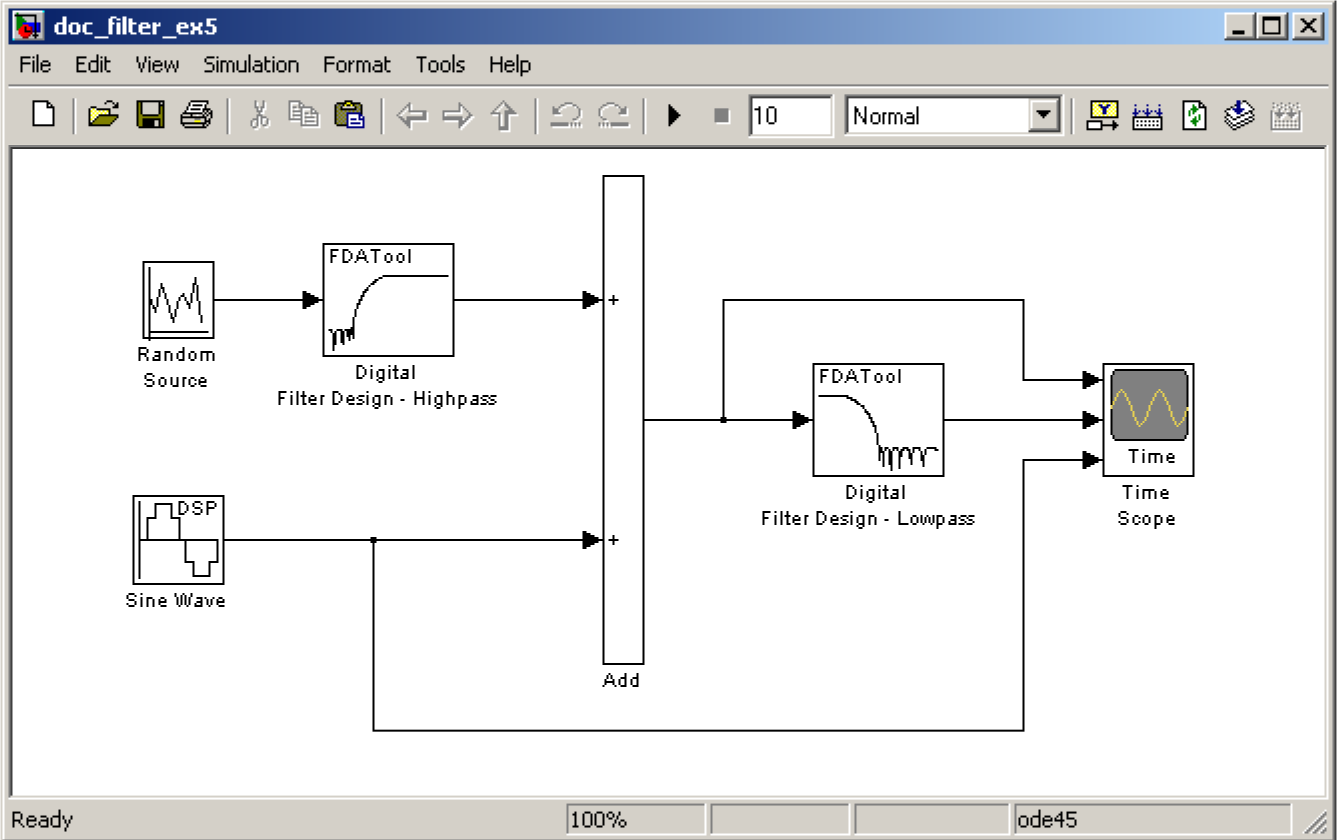
Block	Library	Quantity
Add	Simulink Math Operations library	1
Random Source	Signal Processing Sources	1
Sine Wave	Signal Processing Sources	1
Time Scope	Signal Processing Sinks	1

3 Set the parameters for these blocks as indicated in the following table. Leave the parameters not listed in the table at their default settings.

Parameter Settings for the Other Blocks

Block	Parameter Setting
Add	• **Icon shape** = `rectangular` • **List of signs** = `++`
Random Source	• **Source type =** = `Uniform` • **Minimum** = `0` • **Maximum** = `4` • **Sample mode** = `Discrete` • **Sample time** = `1/1000` • **Samples per frame** = `50`
Sine Wave	• **Frequency (Hz)** = `75` • **Sample time** = `1/1000` • **Samples per frame** = `50`
Time Scope	• **File > Number of Input Ports > 3** • **File > Configuration ...** – Open the **Visuals:Time Domain Options** dialog and set **Time range** = `Input sample time`

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 current numbered channel names with the following:

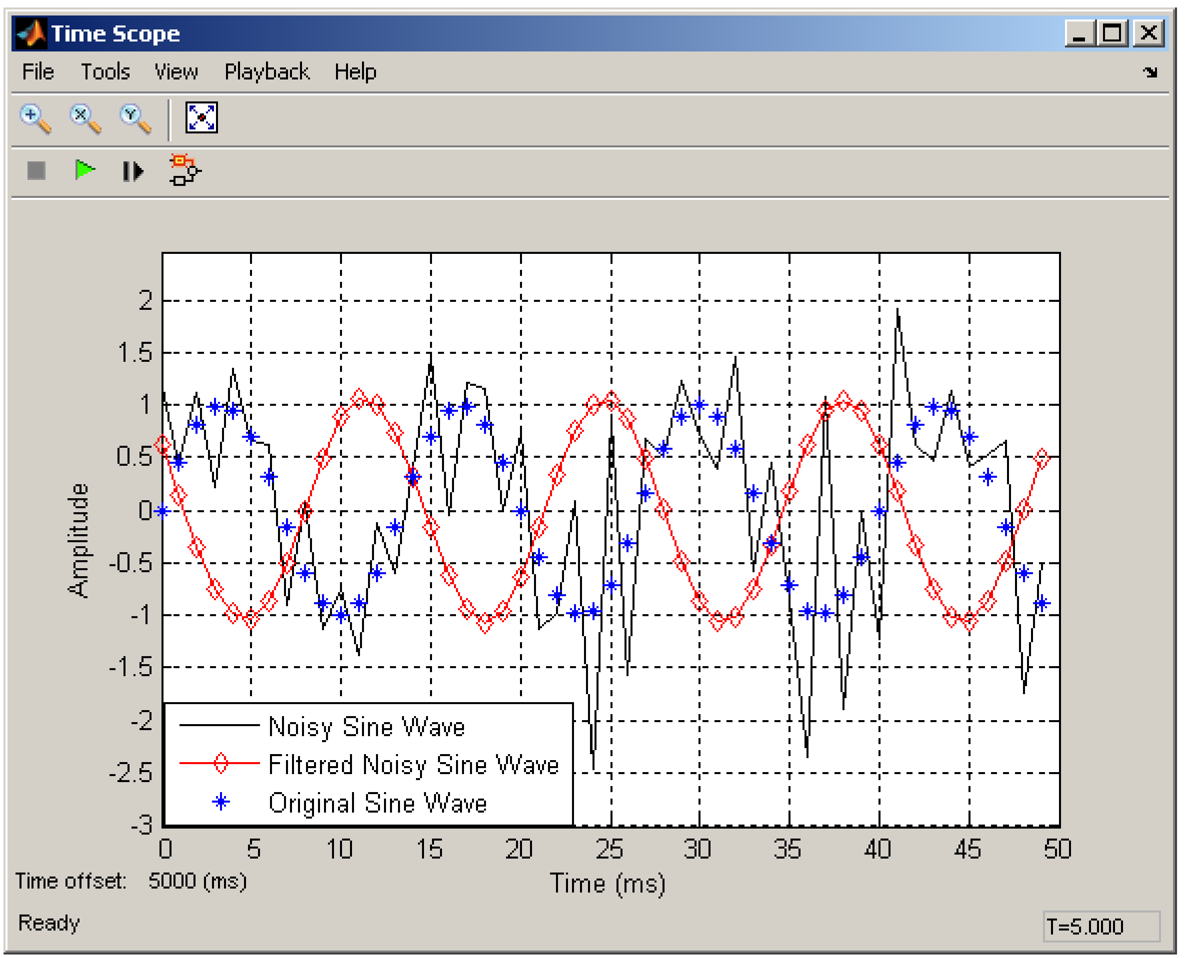
   - Channel 1 = `Noisy Sine Wave`
   - Channel 2 = `Filtered Noisy Sine Wave`
   - Channel 3 = `Original Sine Wave`

   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 3-31. 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 "Importing and Exporting Quantized Filters" in the Filter Design Toolbox documentation.

Filter Realization Wizard

In this section...
"Overview of the Filter Realization Wizard" on page 3-31
"Designing and Implementing a Fixed-Point Filter" on page 3-32
"Setting the Filter Structure and Number of Filter Sections" on page 3-48
"Optimizing the Filter Structure" on page 3-49

Overview of the Filter Realization Wizard

The Filter Realization Wizard is another Signal Processing Blockset 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 Signal Processing Blockset 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 "Choosing Between Filter Design Blocks" on page 3-18.

Alternatively, you can use other MathWorks products, such as Signal Processing Toolbox software and Filter Design Toolbox software, to design your filters. Once you design a filter using either toolbox, you can use one of the filter implementation blocks from Signal Processing Blockset software, such as the Digital Filter block, to realize the filters in your models. For more information, see the Signal Processing Toolbox documentation and Filter Design Toolbox documentation. To learn how to import and export your filter designs, see "Importing and Exporting Quantized Filters" in the Filter Design Toolbox documentation.

Designing and Implementing a Fixed-Point Filter

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 — Creating a Signal with Added Noise" on page 3-32
- "Part 2 — Creating a Fixed-Point Filter with the Filter Realization Wizard" on page 3-34
- "Part 3 — Building a Model to Filter a Signal" on page 3-42
- "Part 4 — Looking at Filtering Results" on page 3-46

Part 1 — Creating a Signal with Added Noise

In this section of the tutorial, you will create a signal with added noise. Later in the tutorial, you will filter this signal with a fixed-point filter that you design with the Filter Realization Wizard.

1 Type

```
load mtlb
soundsc(mtlb,Fs)
```

at the MATLAB command line. You should hear a voice say "MATLAB." This is the signal to which you will add noise.

2 Create a noise signal by typing

```
noise = cos(2*pi*3*Fs/8*(0:length(mtlb)-1)/Fs)';
```

at the command line. You can hear the noise signal by typing

```
soundsc(noise,Fs)
```

3 Add the noise to the original signal by typing

```
u = mtlb + noise;
```

at the command line.

4 Scale the signal with noise by typing

```
u = u/max(abs(u));
```

at the command line. You scale the signal to try to avoid overflows later on. You can hear the scaled signal with noise by typing

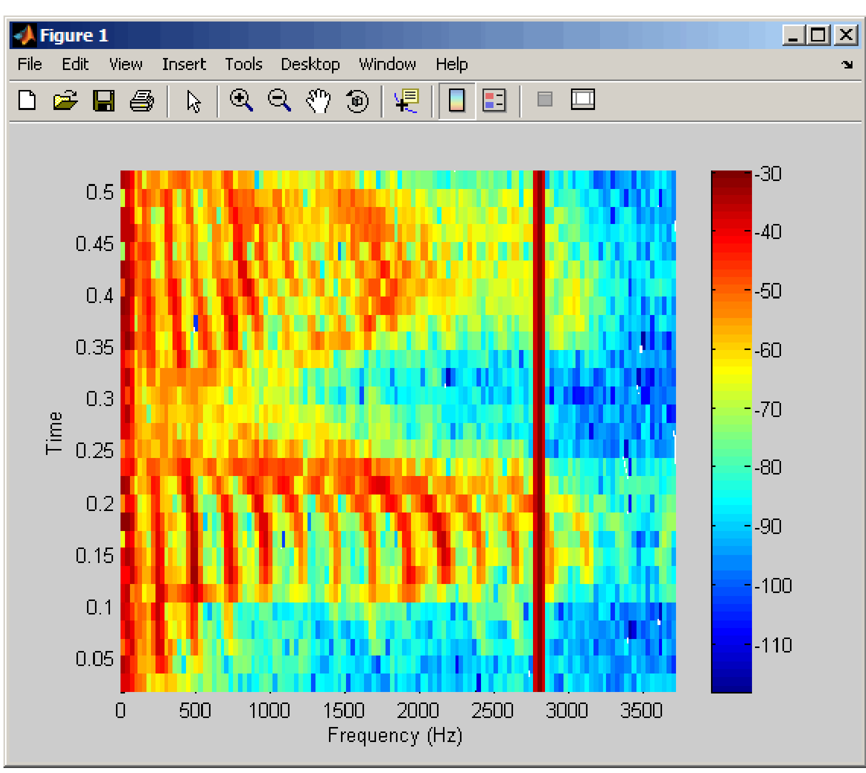
```
soundsc(u,Fs)
```

5 View the scaled signal with noise by typing

```
spectrogram(u,256,[],[],Fs);colorbar
```

at the command line.

The spectrogram appears as follows.

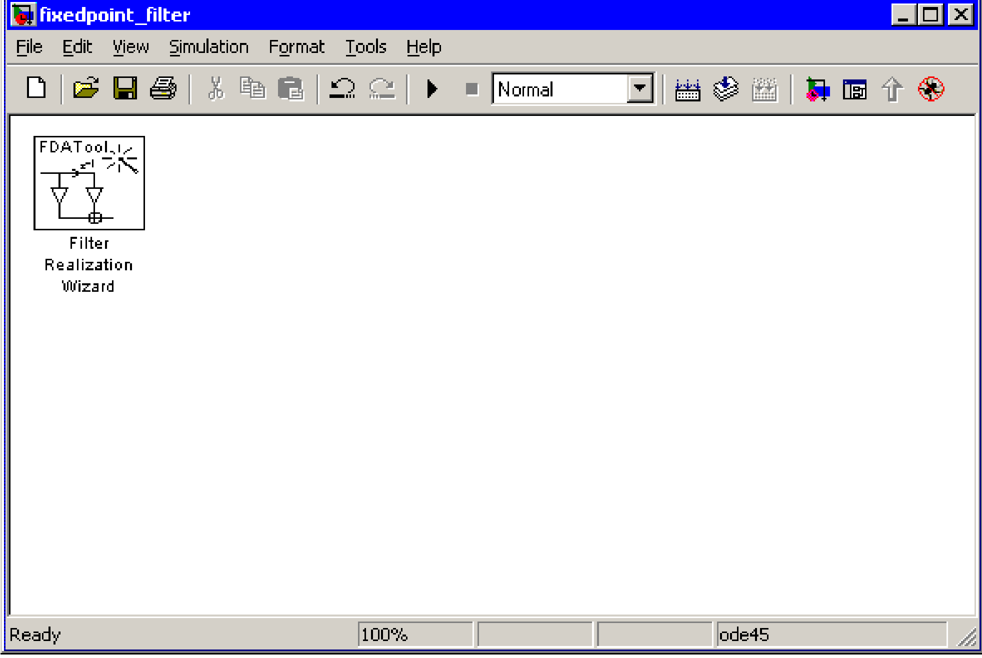


In the spectrogram, you can see the noise signal as a line at about 2800 Hz, which is equal to `3*Fs/8`.

Part 2 — Creating 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.

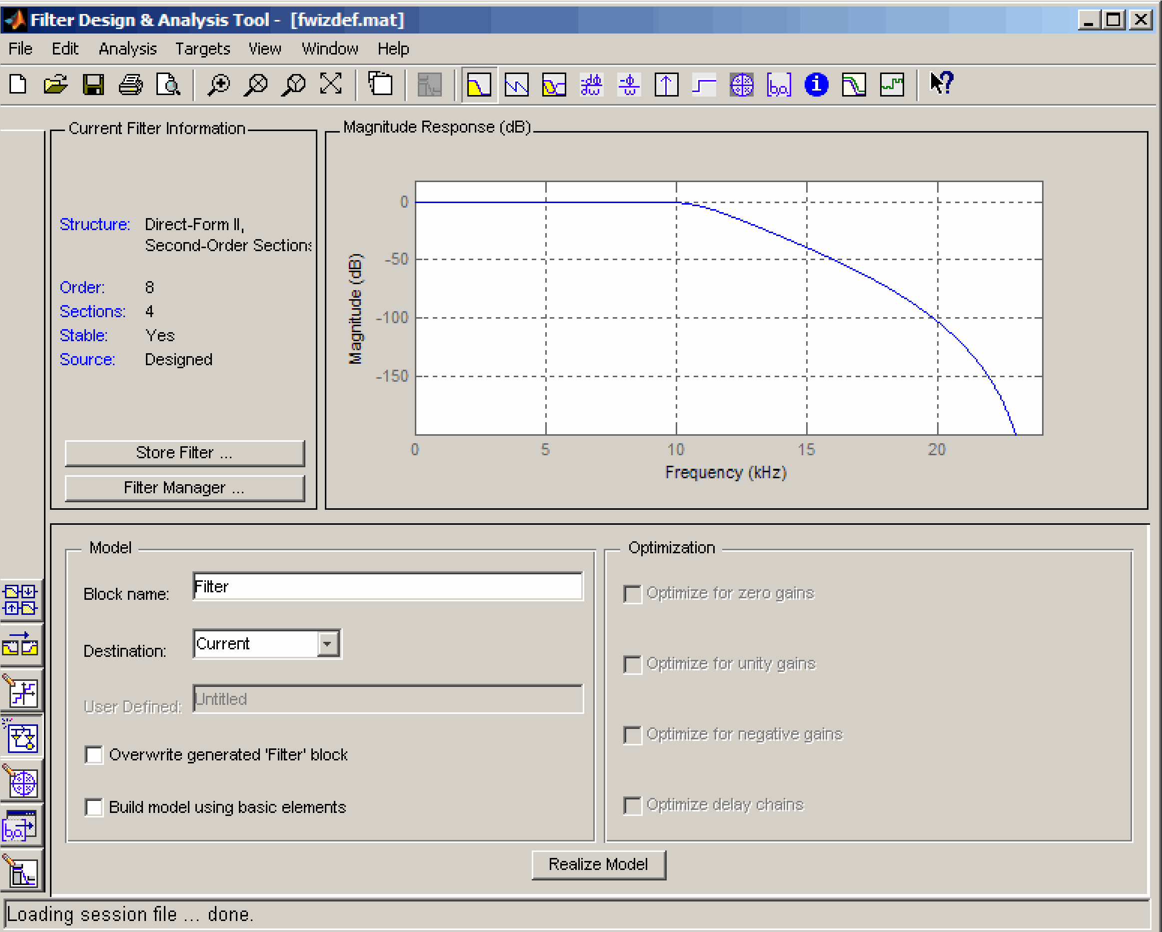
6 Open a new Simulink model, and drag-and-drop a Filter Realization Wizard block from the Filtering / Filter Implementations library into the model.



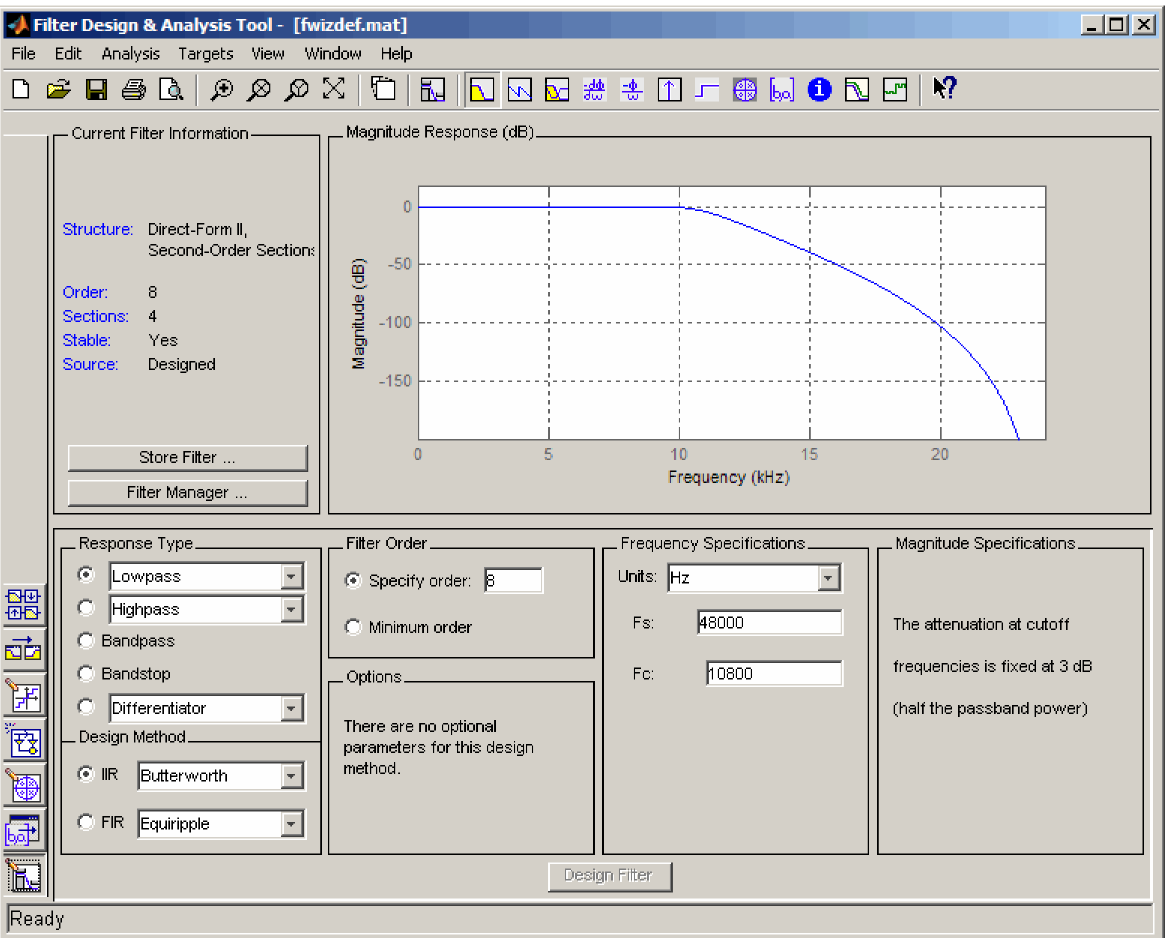
Note You do not have to place a Filter Realization Wizard block in a model in order to use it. You can open the GUI from within a library. However, for purposes of this tutorial, we will keep the Filter Realization Wizard block in the model.

---

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.

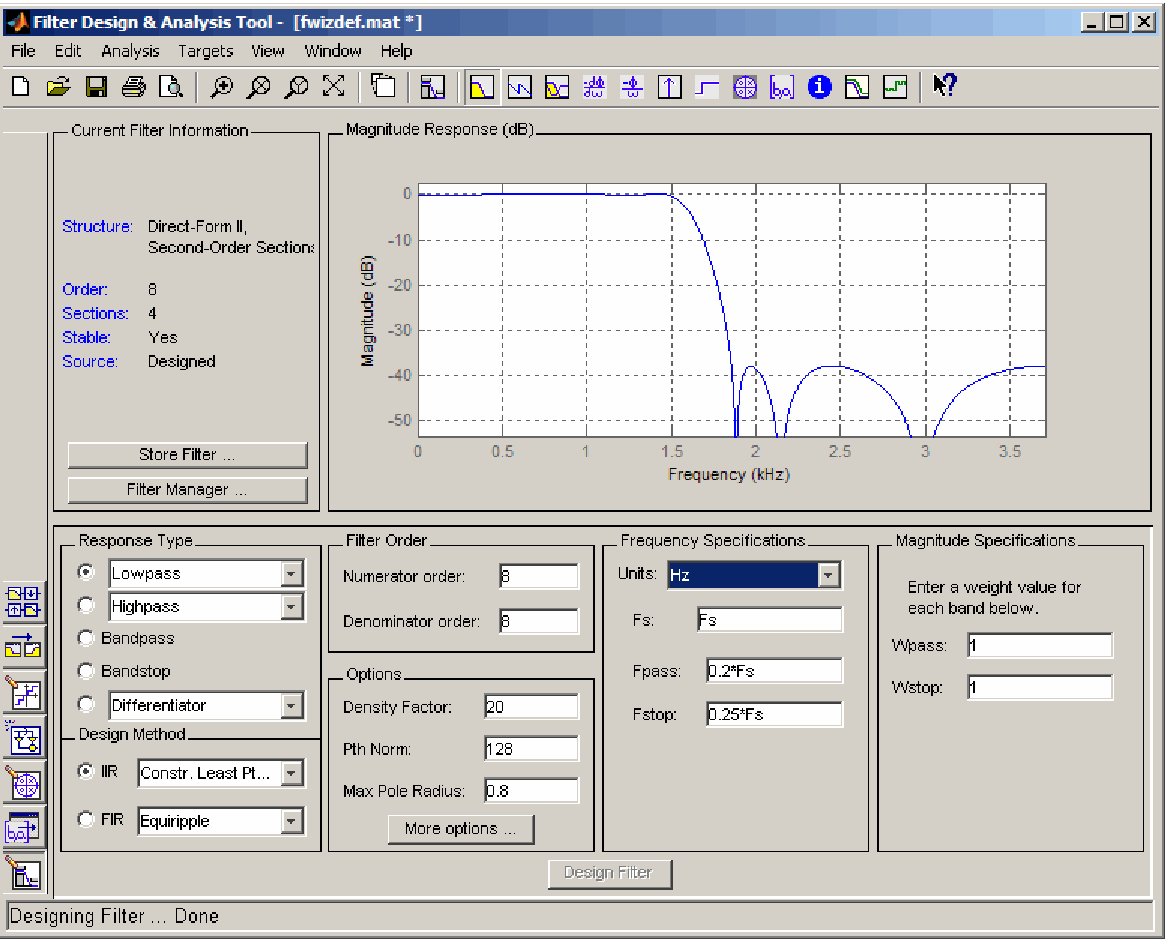


9 Set the following fields in the **Design Filter** panel:

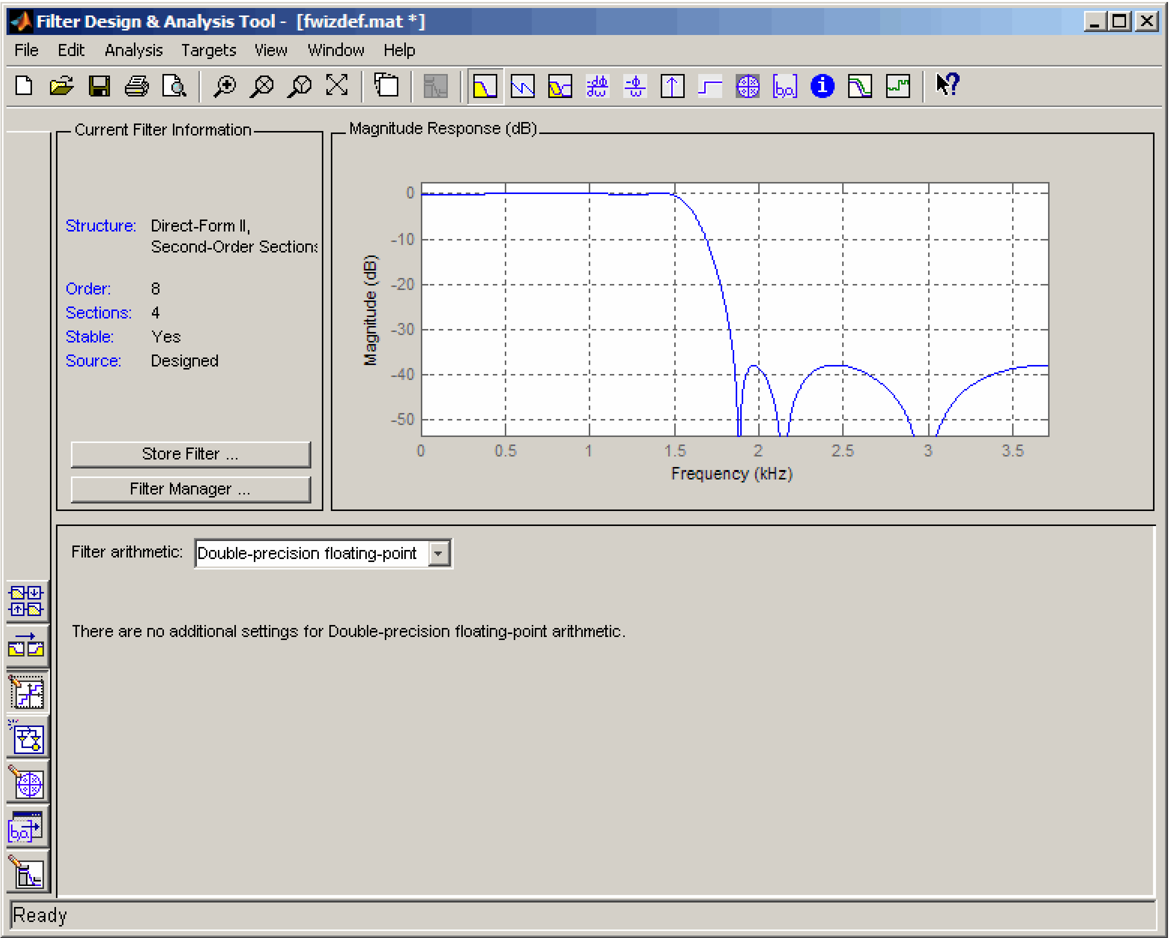
- Set **Design Method** to `IIR -- Constrained Least Pth-norm`
- Set **Fs** to `Fs`
- Set **Fpass** to `0.2*Fs`
- Set **Fstop** to `0.25*Fs`

- Set **Max pole radius** to 0.8
- Click the **Design Filter** button

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



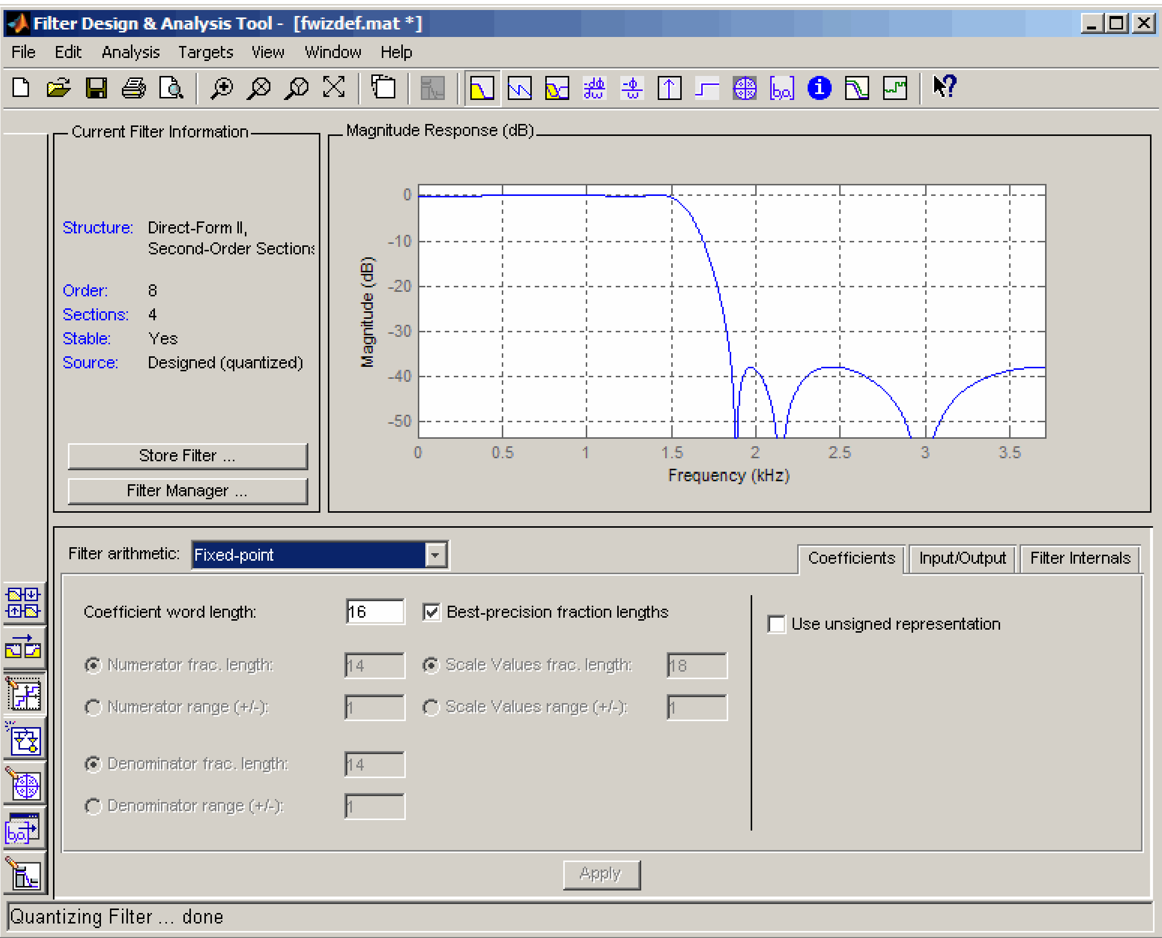
10 Click the **Set Quantization Parameters** button on the bottom left of FDATool. This brings forward the **Set Quantization Parameters** panel of the tool.



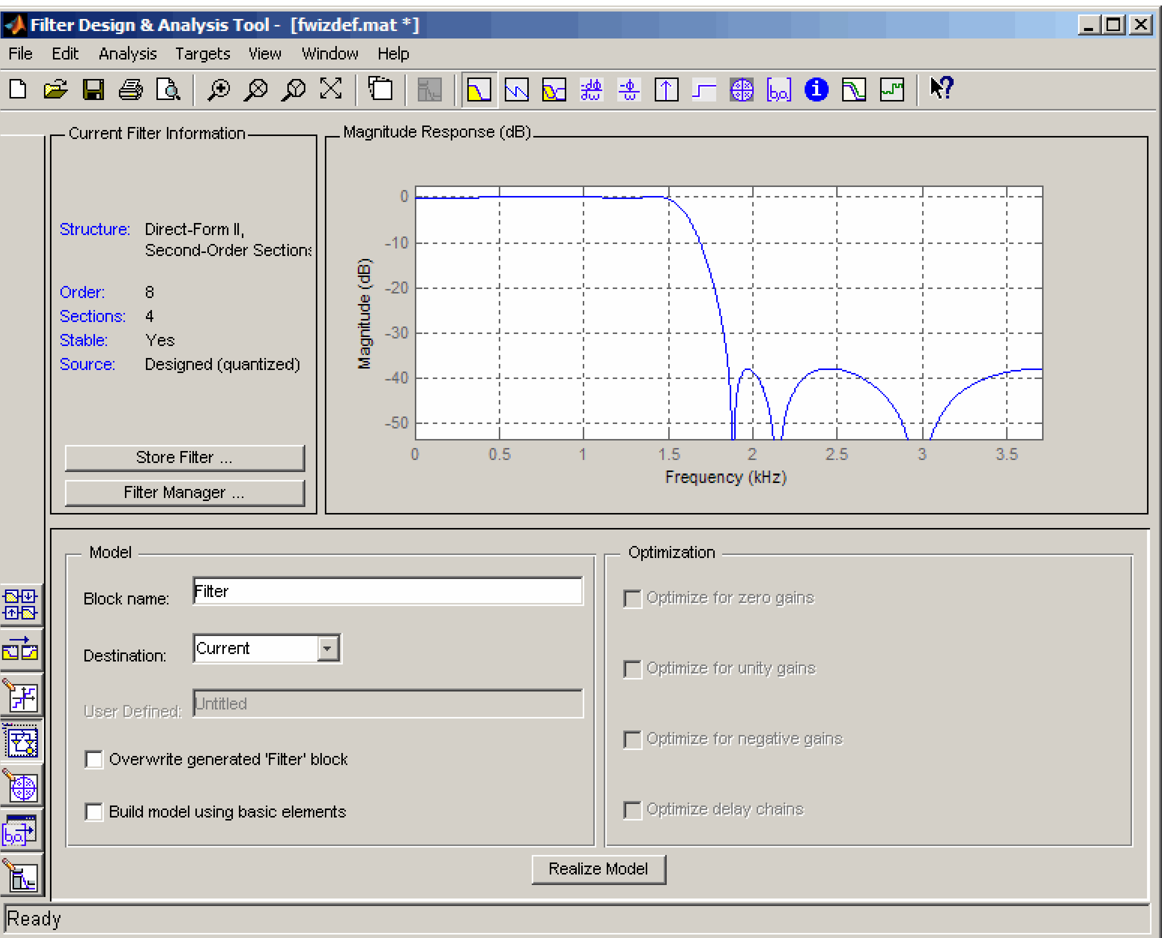
11 Set the following fields in the **Set Quantization Parameters** panel:

- Select `Fixed-point` for the **Filter arithmetic** parameter.
- Make sure the **Best precision fraction lengths** check box is selected on the **Coefficients** pane.

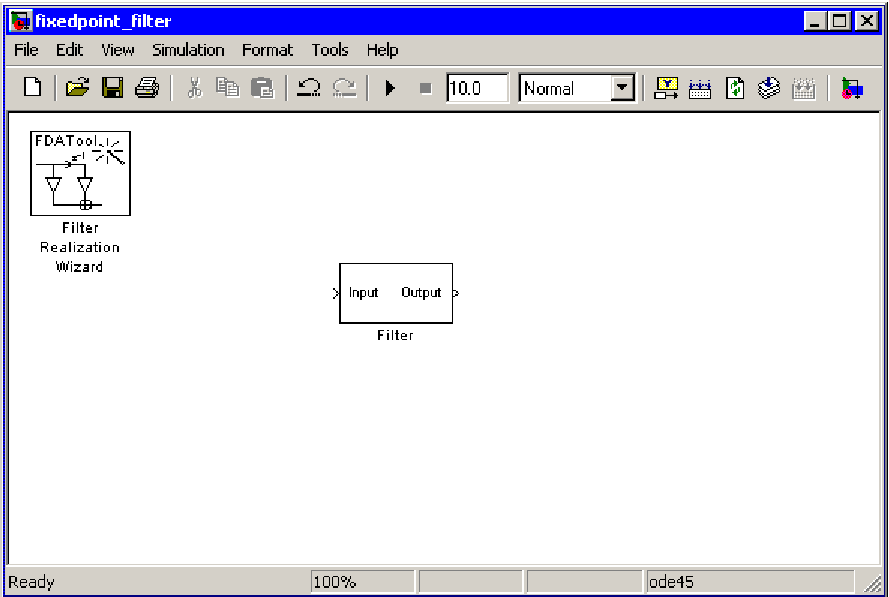
The **Set Quantization Parameters** panel should appear as follows.



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



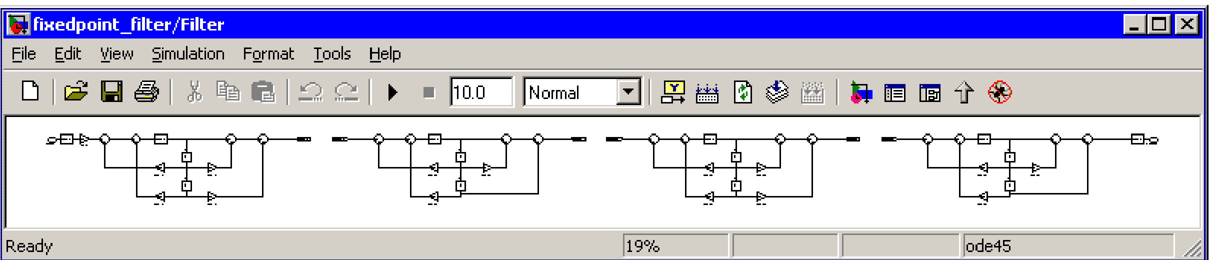
13 Select the **Build model using basic elements** check box, then click the **Realize Model** button on the bottom of FDATool. A block for the new filter appears in your model.



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

---

14 Double-click the Filter block in your model. This will bring up the realization of the filter being represented by the block.

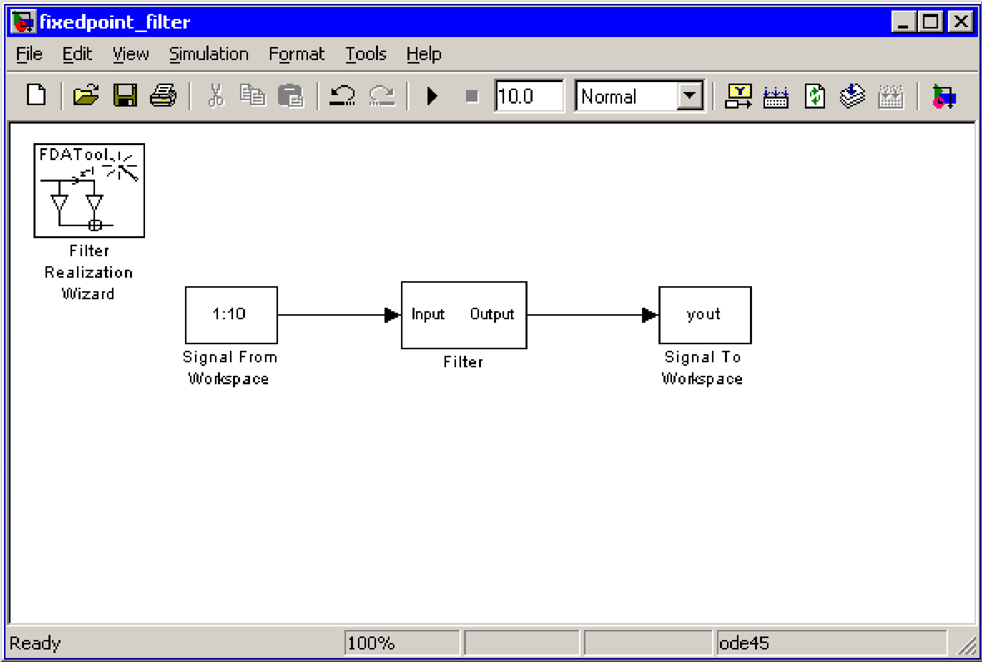


Part 3 — Building a Model to Filter a Signal

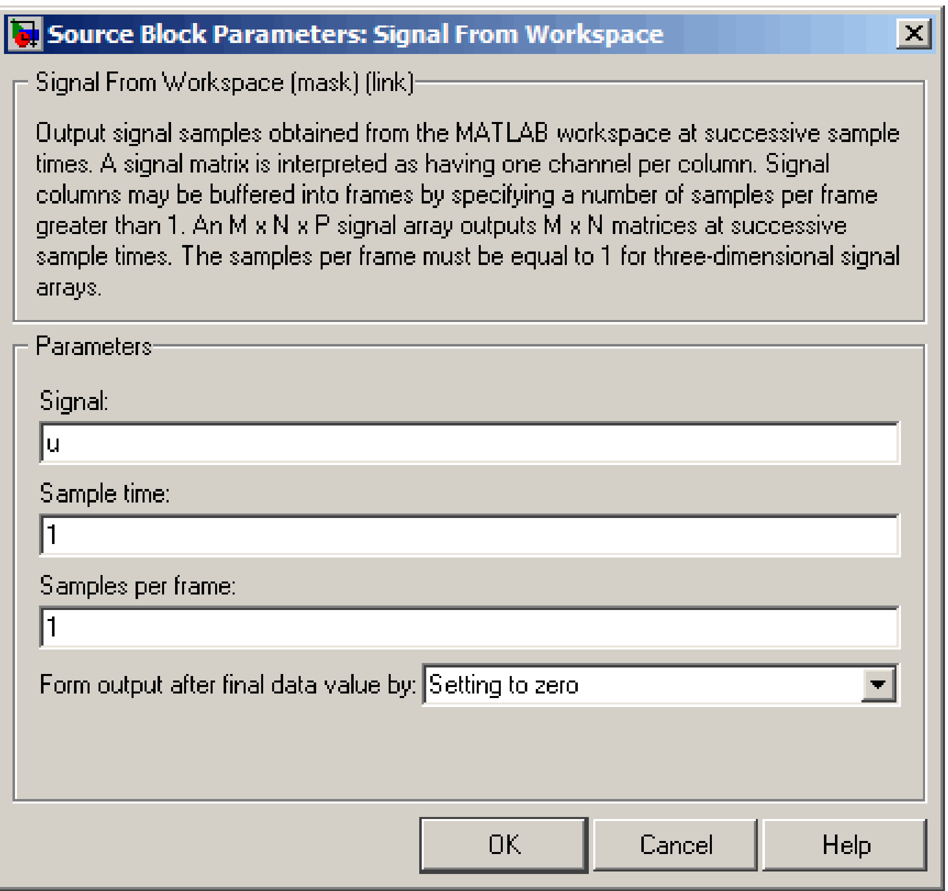
In this section of the tutorial, you will build and run a model with the filter you just designed, in order to filter the noise from your signal.

15 Connect a Signal From Workspace block from the Signal Processing Sources library to the input port of your filter block.

16 Connect a Signal To Workspace block from the Signal Processing Sinks library to the output port of your filter block. Your model should now appear as follows.



17 Change the **Signal** parameter of the Signal From Workspace block to u by double-clicking on the block.

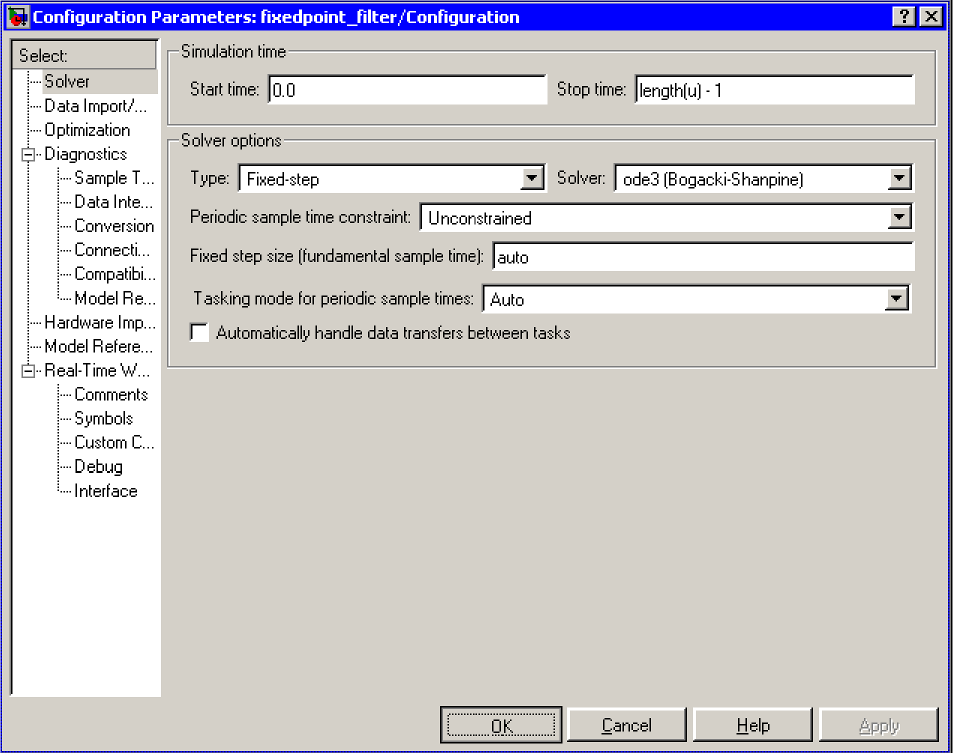


18 Click the **OK** button.

19 Open the **Configuration Parameters** dialog box from the **Simulation** menu of the model. In the **Solver** pane of the dialog, set the following fields:

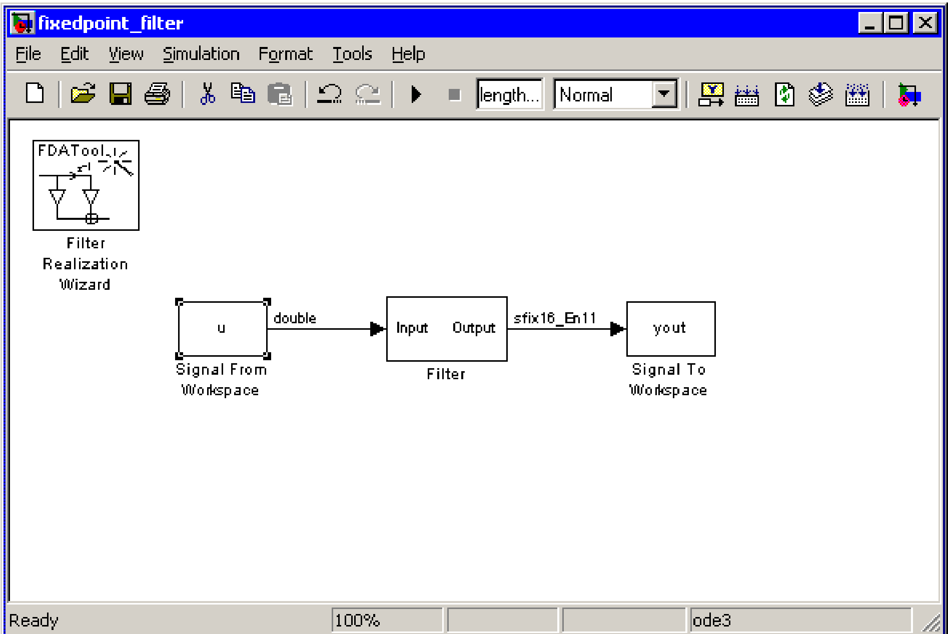
- **Stop time** = `length(u)-1`
- **Type** = `Fixed-step`

The **Configuration Parameters** dialog box should now appear as follows.



20 Click the **OK** button.

21 Run the model.



22 Select **Port/Signal Displays** > **Port Data Types** from the **Format** menu. You can you see that a signal of type `double` is entering your Filter block, and a signal of type `sfix16_En11` is exiting your Filter block.

Part 4 — Looking at Filtering Results

Now you can listen to and look at the results of the fixed-point filter you designed and implemented.

23 Type

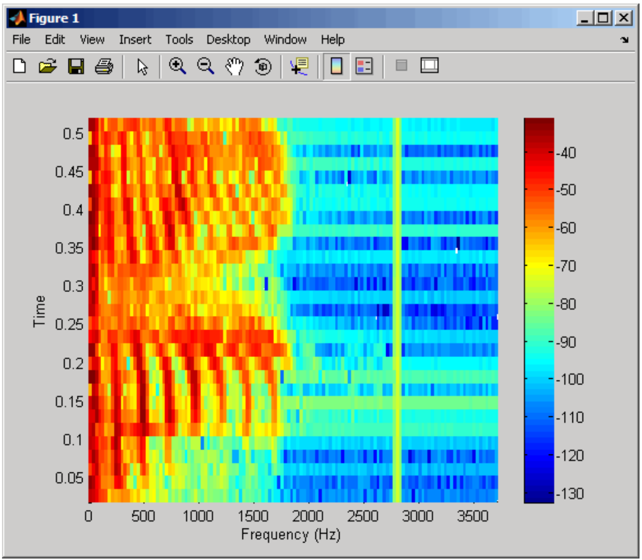
```
soundsc(yout,Fs)
```

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.

24 Type

```
figure
spectrogram(yout,256,[],[],Fs);colorbar
```

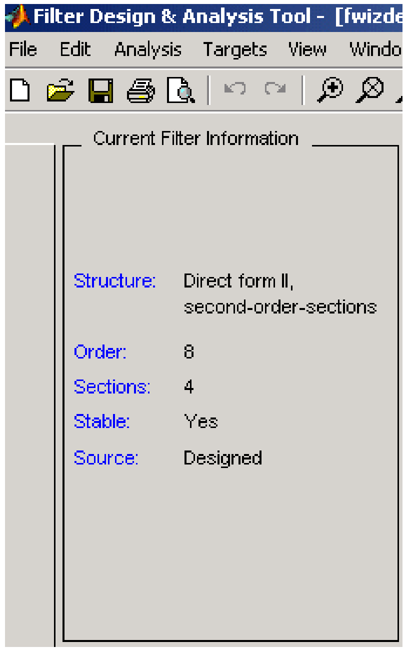
at the command line.



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.

Setting 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.

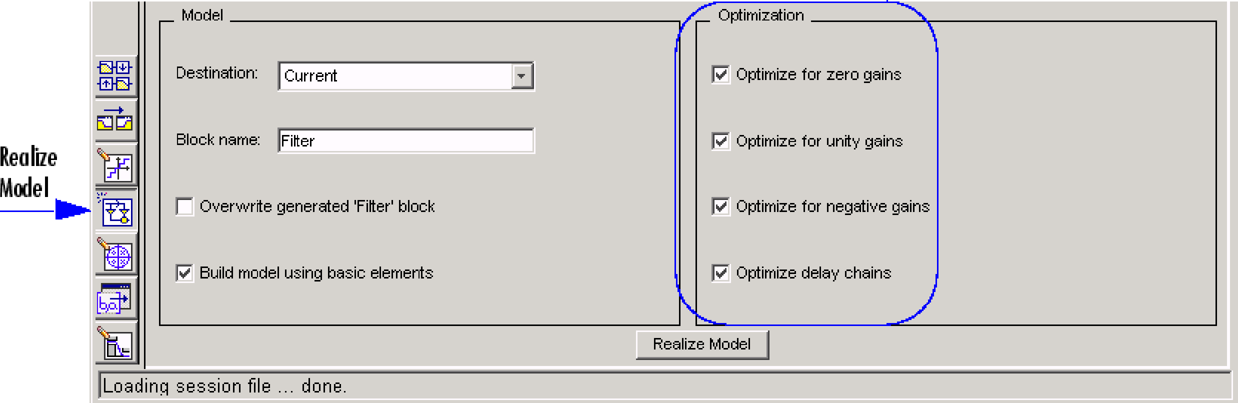
Note You might not be able to directly access some of the supported structures through the **Convert Structure** dialog of FDATool. However, you *can* access all of the structures by creating a `dfilt` filter object with the desired structure, and then importing the filter into FDATool. To learn more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

Optimizing 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.

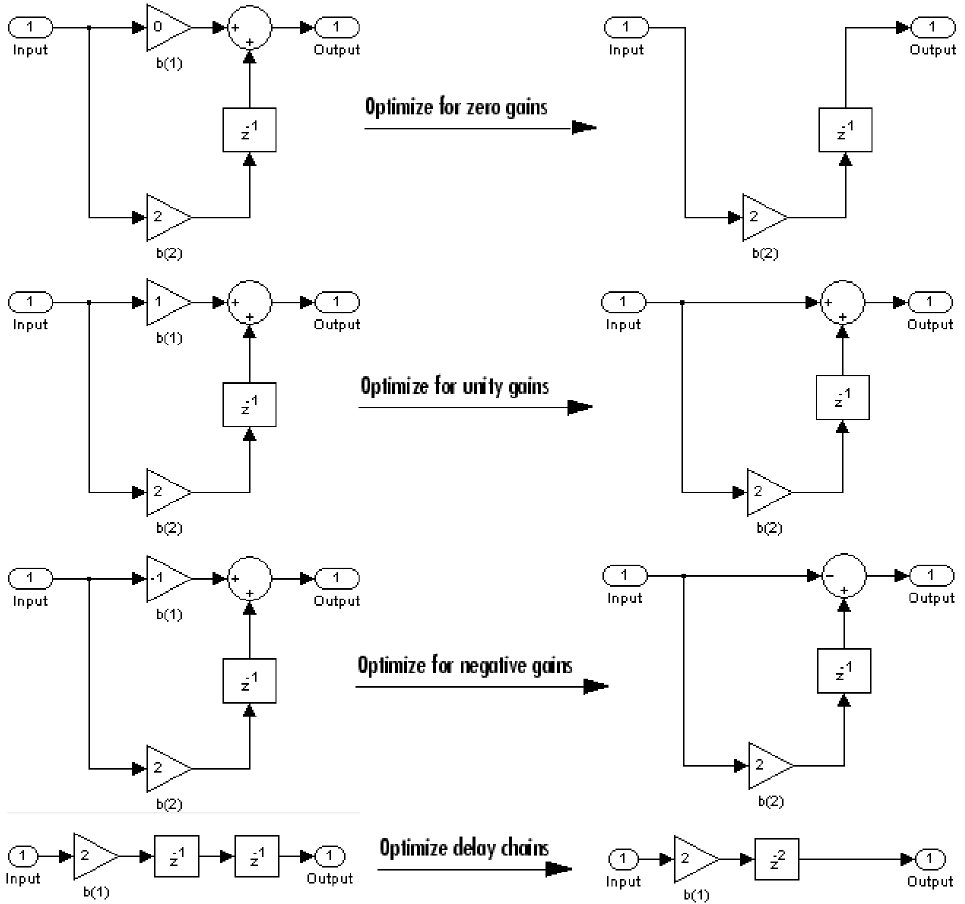
2. Select the desired optimizations in the **Optimization** region of the **Realize Model** pane. See the following descriptions and illustrations of each optimization option.



- **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*.

The following diagram illustrates the results of each of these optimizations.



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}	Ω_{p1}, Ω_{p2}, R_p	Ω_{s1}, Ω_{s2}, R_s	Ω_{p1}, Ω_{p2}, R_p, R_s
Bandstop	Ω_{p1}, Ω_{p2}	Ω_{p1}, Ω_{p2}, R_p	Ω_{s1}, Ω_{s2}, R_s	Ω_{p1}, Ω_{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.

Adaptive Filters

In this section...

"Creating an Acoustic Environment" on page 3-53
"Creating an Adaptive Filter" on page 3-55
"Customizing an Adaptive Filter" on page 3-60
"Adaptive Filtering Demos" on page 3-64

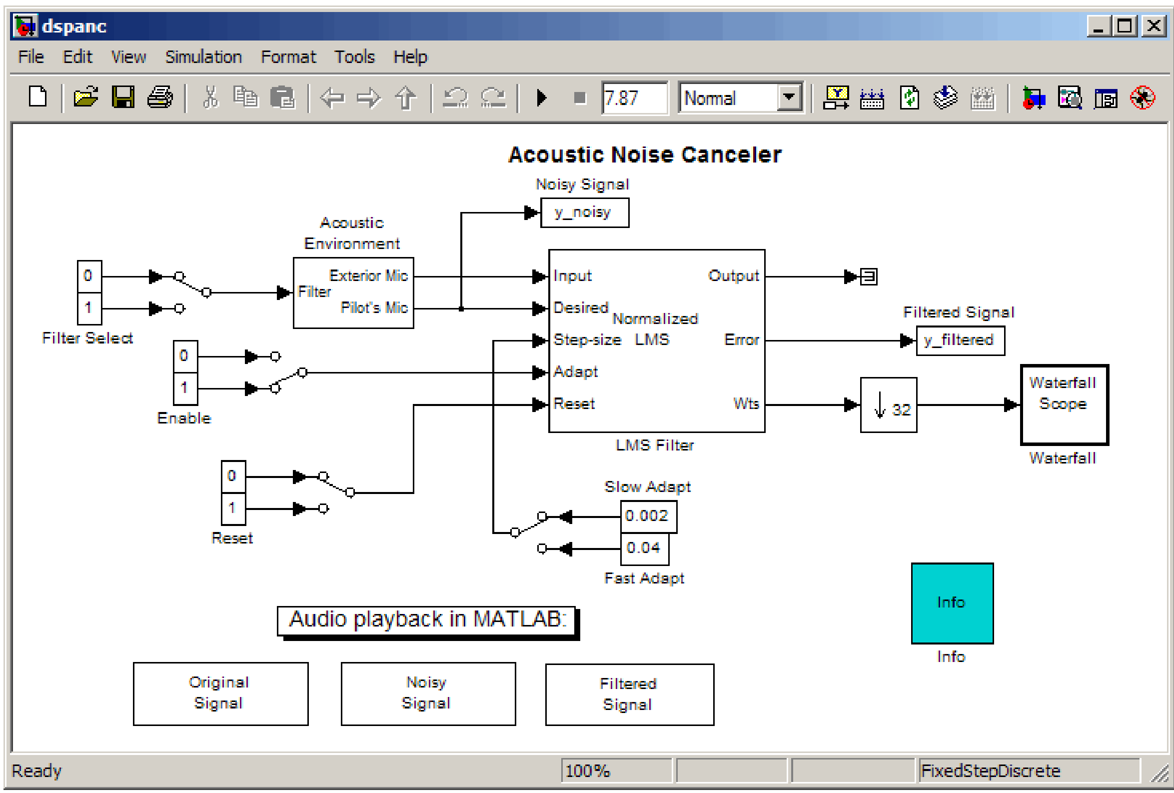
Creating an Acoustic Environment

Adaptive filters are filters whose coefficients or weights change over time to adapt to the statistics of a signal. They are used in a variety of fields including communications, controls, radar, sonar, seismology, and biomedical engineering.

In this topic, you learn how to create an acoustic environment that simulates both white noise and colored noise added to an input signal. You later use this environment to build a model capable of adaptive noise cancellation using adaptive filtering:

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

The Signal Processing Blockset 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.

Creating an Adaptive Filter

In the previous topic, “Creating an Acoustic Environment” on page 3-53, 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 “Creating an Acoustic Environment” on page 3-53 is not open on your desktop, you can open an equivalent model by typing

   ```
   doc_adapt1_audio
   ```

   at the MATLAB command prompt.

2. From the Signal Processing Blockset Filtering library, and then from the Adaptive Filters library, click-and-drag an LMS Filter block into the model that contains the Acoustic Environment subsystem.

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

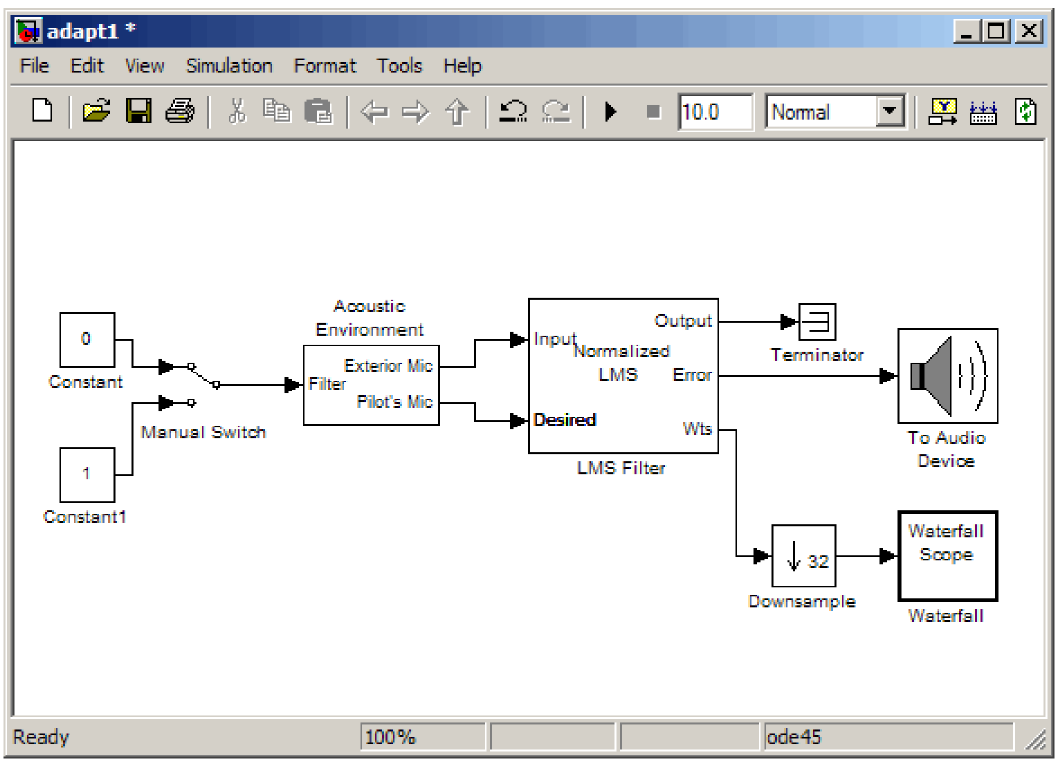
   - **Algorithm** = `Normalized LMS`
   - **Filter length** = `40`
   - **Step size (mu)** = `0.002`
   - **Leakage factor (0 to 1)** = `1`

   The block uses the normalized LMS algorithm to calculate the forty filter coefficients. Setting the **Leakage factor (0 to 1)** parameter to `1` means that the current filter coefficient values depend on the filter’s initial conditions and all of the previous input values.

4. Click-and-drag the following blocks into your model.

Block	Library	Quantity
Constant	Simulink/Sources	2
Manual Switch	Simulink/Signal Routing	1
Terminator	Simulink/Sinks	1
Downsample	Signal Operations	1
To Audio Device	Signal Processing Sinks	1
Waterfall Scope	Signal Processing Sinks	1

5 Connect the blocks so that your model resembles the following figure.



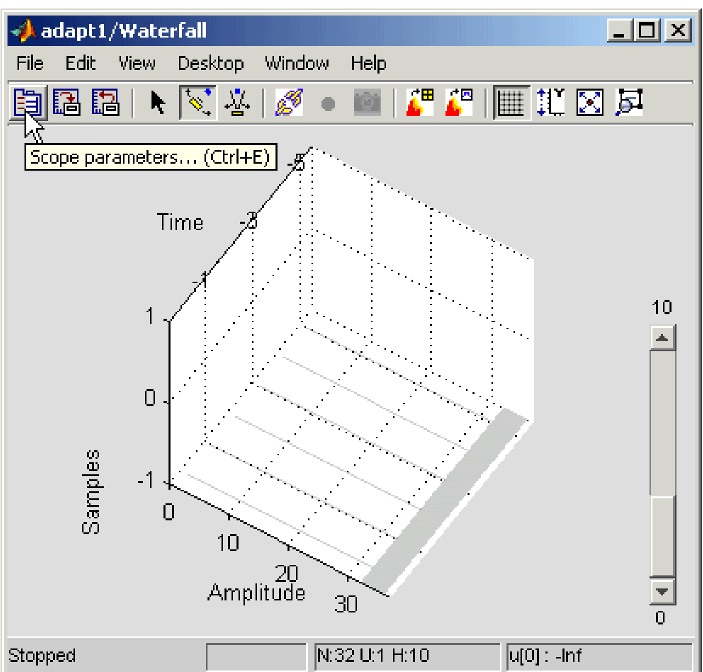
6 Double-click the Constant block. Set the **Constant value** parameter to `0` and then click **OK**.

7 Double-click the Downsample block. Set the **Downsample factor, K** parameter to 32. Click **OK**.

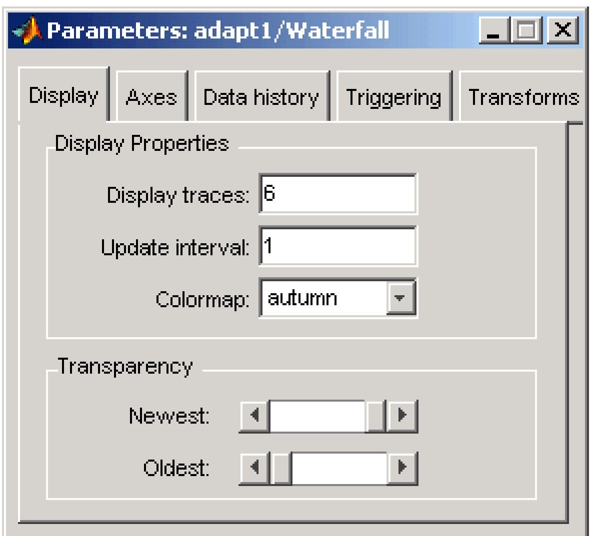
The filter weights are being updated so often that there is very little change from one update to the next. To see a more noticeable change, you need to downsample the output from the Wts port.

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`

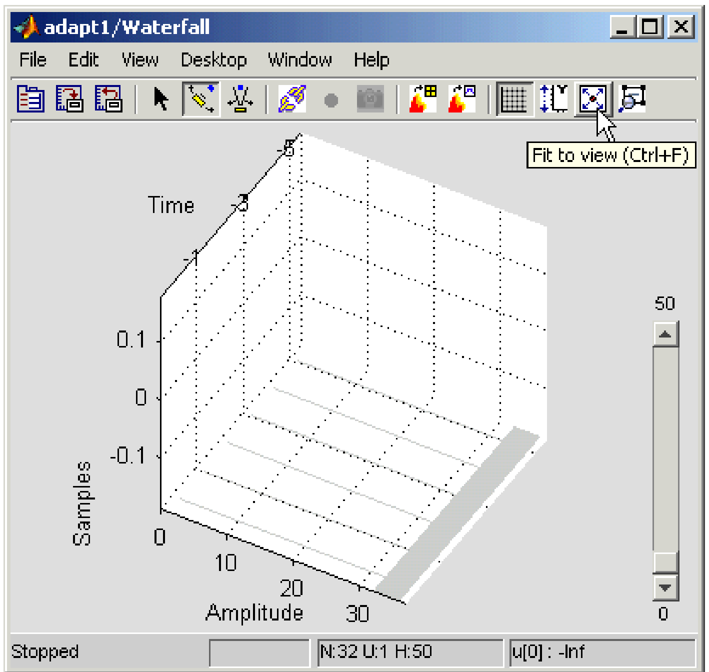
11 Click the **Data history** tab. Set the parameters as follows:

- **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, "Customizing an Adaptive Filter" on page 3-60, you modify the LMS Filter block and change its parameters during simulation.

Customizing an Adaptive Filter

In the previous topic, "Creating an Adaptive Filter" on page 3-55, 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 "Creating an Acoustic Environment" on page 3-53 is not open on your desktop, you can open an equivalent model by typing

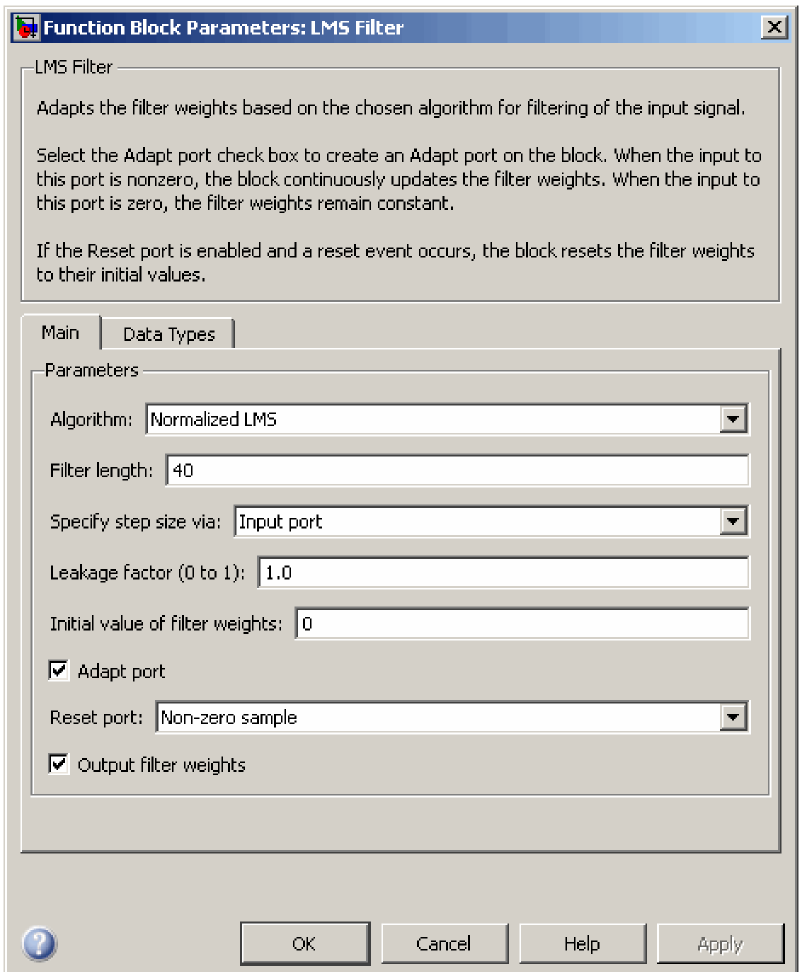
   ```
   doc_adapt2_audio
   ```

   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.

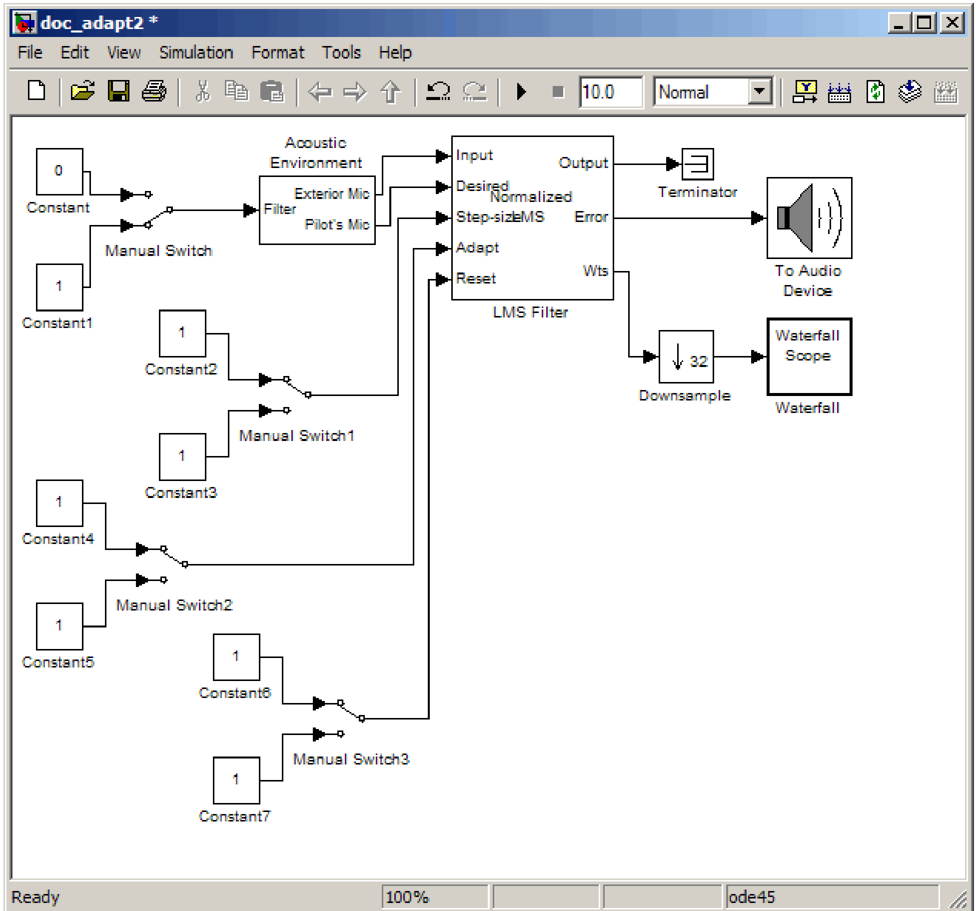


Step-size, Adapt, and Reset ports appear on the LMS Filter block.

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 to 0.

   The filter weights displayed in the **Waterfall** scope window remain constant. When the input to the Adapt port is 0, the filter weights are not updated.

13 Double-click Manual Switch2 so that the input to the Adapt port is 1.

   The LMS Filter block updates the coefficients.

14 Connect the Manual Switch1 block to the Constant block that represents 0.002. Then, change the input to the Acoustic Environment subsystem. Repeat this procedure with the Constant block that represents 0.04.

   You can see that the system reaches steady state faster when the step size is larger.

15 Double-click the Manual Switch3 block so that the input to the Reset port is 1.

   The block resets the filter weights to their initial values. In the **Block Parameters: LMS Filter** dialog box, from the **Reset port** list, you chose `Non-zero sample`. This means that any nonzero input to the Reset port triggers a reset operation.

You have now experimented with adaptive noise cancellation using the LMS Filter block. You adjusted the parameters of your adaptive filter and viewed the effects of your changes while the model was running.

For more information about adaptive filters, see the following block reference pages:

- LMS Filter
- RLS Filter
- Block LMS Filter
- Fast Block LMS Filter

Adaptive Filtering Demos

Signal Processing Blockset 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	`lmsadeq`
LMS Adaptive Time-Delay Estimation	`lmsadtde`
Nonstationary Channel Estimation	`kalmnsce`
RLS Adaptive Noise Cancellation	`rlsdemo`

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 Signal Processing Blockset demos, type `demo blockset signal` at the MATLAB command line.

Multirate Filters

In this section...

"Filter Banks" on page 3-66

"Multirate Filtering Examples" on page 3-74

Filter Banks

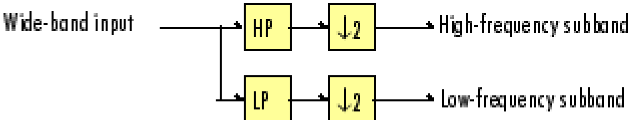
Multirate filters alter the sample rate of the input signal during the filtering process. Such filters are useful in both rate conversion and filter bank applications.

The Dyadic Analysis Filter Bank block decomposes a broadband signal into a collection of subbands with smaller bandwidths and slower sample rates. The Dyadic Synthesis Filter Bank block reconstructs a signal decomposed by the Dyadic Analysis Filter Bank block.

To use a dyadic synthesis filter bank to perfectly reconstruct the output of a dyadic analysis filter bank, the number of levels and tree structures of both filter banks *must* be the same. In addition, the filters in the synthesis filter bank *must* be designed to perfectly reconstruct the outputs of the analysis filter bank. Otherwise, the reconstruction will not be perfect.

Dyadic Analysis Filter Banks

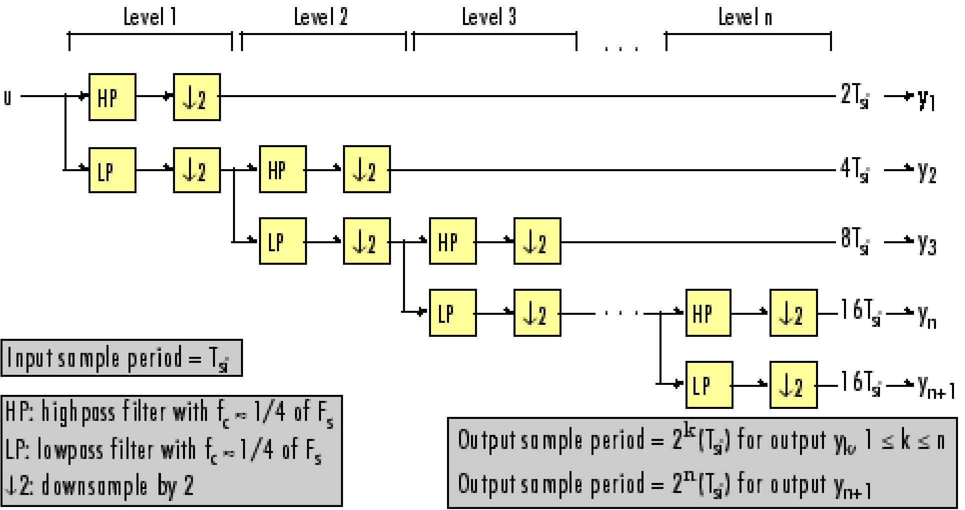
Dyadic analysis filter banks are constructed from the following basic unit. The unit can be cascaded to construct dyadic analysis filter banks with either a symmetric or asymmetric tree structure.



Each unit consists of a lowpass (LP) and highpass (HP) FIR filter pair, followed by a decimation by a factor of 2. The filters are halfband filters with a cutoff frequency of F_s / 4, a quarter of the input sampling frequency. Each filter passes the frequency band that the other filter stops.

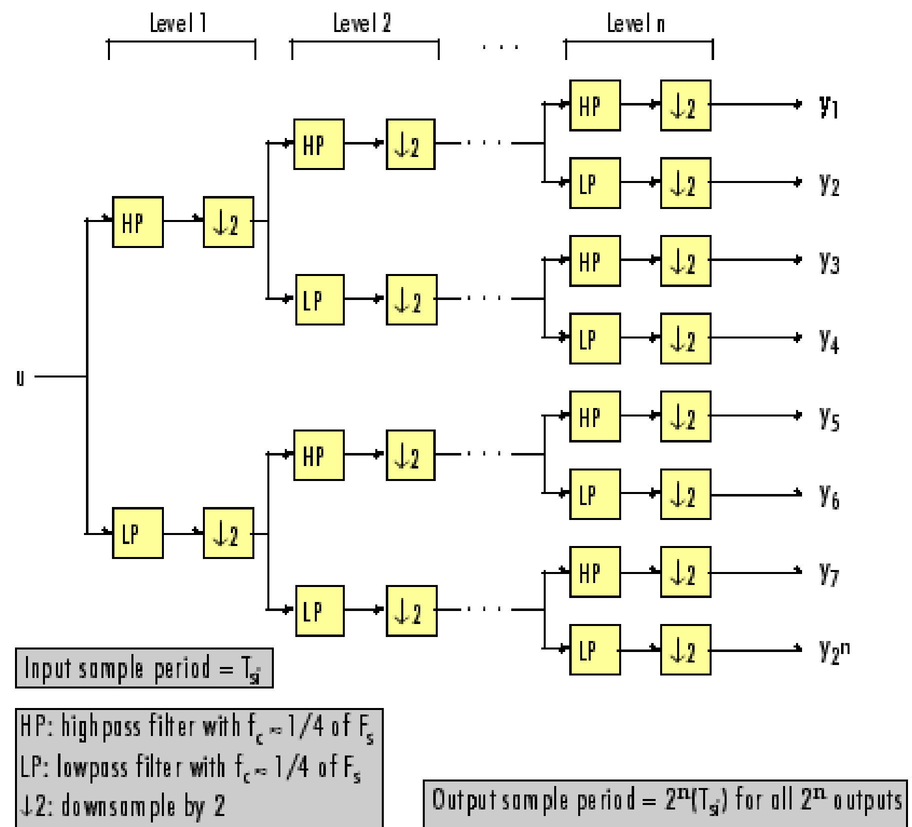
The unit decomposes its input into adjacent high-frequency and low-frequency subbands. Compared to the input, each subband has half the bandwidth (due to the half-band filters) and half the sample rate (due to the decimation by 2).

Note The following figures illustrate the *concept* of a filter bank, but *not* how the block implements a filter bank; the block uses a more efficient polyphase implementation.



n-Level Asymmetric Dyadic Analysis Filter Bank

Use the above figure and the following figure to compare the two tree structures of the dyadic analysis filter bank. Note that the asymmetric structure decomposes only the low-frequency output from each level, while the symmetric structure decomposes the high- and low-frequency subbands output from each level.



n-Level Symmetric Dyadic Analysis Filter Bank

The following table summarizes the key characteristics of the symmetric and asymmetric dyadic analysis filter bank.

Notable Characteristics of Asymmetric and Symmetric Dyadic Analysis Filter Banks

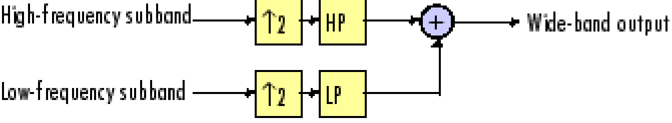
Characteristic	N-Level Symmetric	N-Level Asymmetric
Low- and High-Frequency Subband Decomposition	All the low-frequency and high-frequency subbands in a level are decomposed in the next level.	Each level's low-frequency subband is decomposed in the next level, and each level's high-frequency band is an output of the filter bank.
Number of Output Subbands	2^n	n+1
Bandwidth and Number of Samples in Output Subbands	For an input with bandwidth *BW* and *N* samples, all outputs have bandwidth $BW / 2^n$ and $N / 2^n$ samples.	For an input with bandwidth BW and N samples, y_k has the bandwidth BW_k, and N_k samples, where $BW_k = \begin{cases} BW/2^k & (1 \le k \le n) \\ BW/2^n & (k = n+1) \end{cases}$ $N_k = \begin{cases} N/2^k & (1 \le k \le n) \\ N/2^n & (k = n+1) \end{cases}$ The bandwidth of, and number of samples in each subband (except the last) is half those of the previous subband. The last two subbands have the same bandwidth and number of samples since they originate from the same level in the filter bank.

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 $= \begin{cases} 2^k(T_{si}) & (1 \le k \le n) \\ 2^n(T_{si}) & (k = n+1) \end{cases}$ Due to the decimations by 2, the sample period of each subband (except the last) is twice that of the previous subband. The last two subbands have the same sample period since they originate from the same level in the filter bank.
Total Number of Output Samples	The total number of samples in all of the output subbands is equal to the number of samples in the input (due to the of decimations by 2 at each level).	
Wavelet Applications	In wavelet applications, the highpass and lowpass wavelet-based filters are designed so that the aliasing introduced by the decimations are exactly canceled in reconstruction.	

Dyadic Synthesis Filter Banks

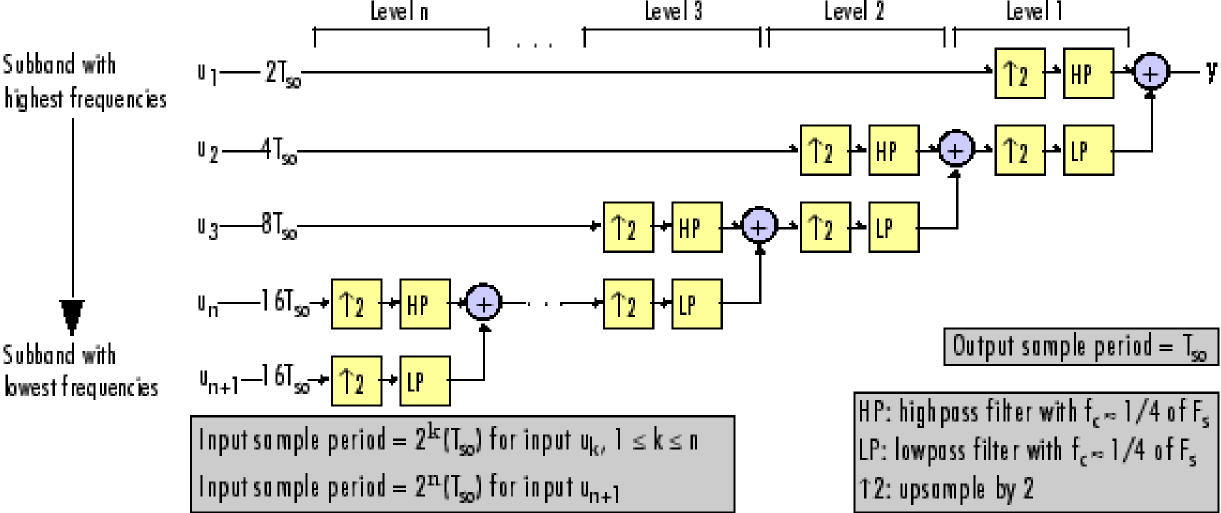
Dyadic synthesis filter banks are constructed from the following basic unit. The unit can be cascaded to construct dyadic synthesis filter banks with either a asymmetric or symmetric tree structure as illustrated in the figures entitled n-Level Asymmetric Dyadic Synthesis Filter Bank and n-Level Symmetric Dyadic Synthesis Filter Bank.



Each unit consists of a lowpass (LP) and highpass (HP) FIR filter pair, preceded by an interpolation by a factor of 2. The filters are halfband filters with a cutoff frequency of F_s / 4, a quarter of the input sampling frequency. Each filter passes the frequency band that the other filter stops.

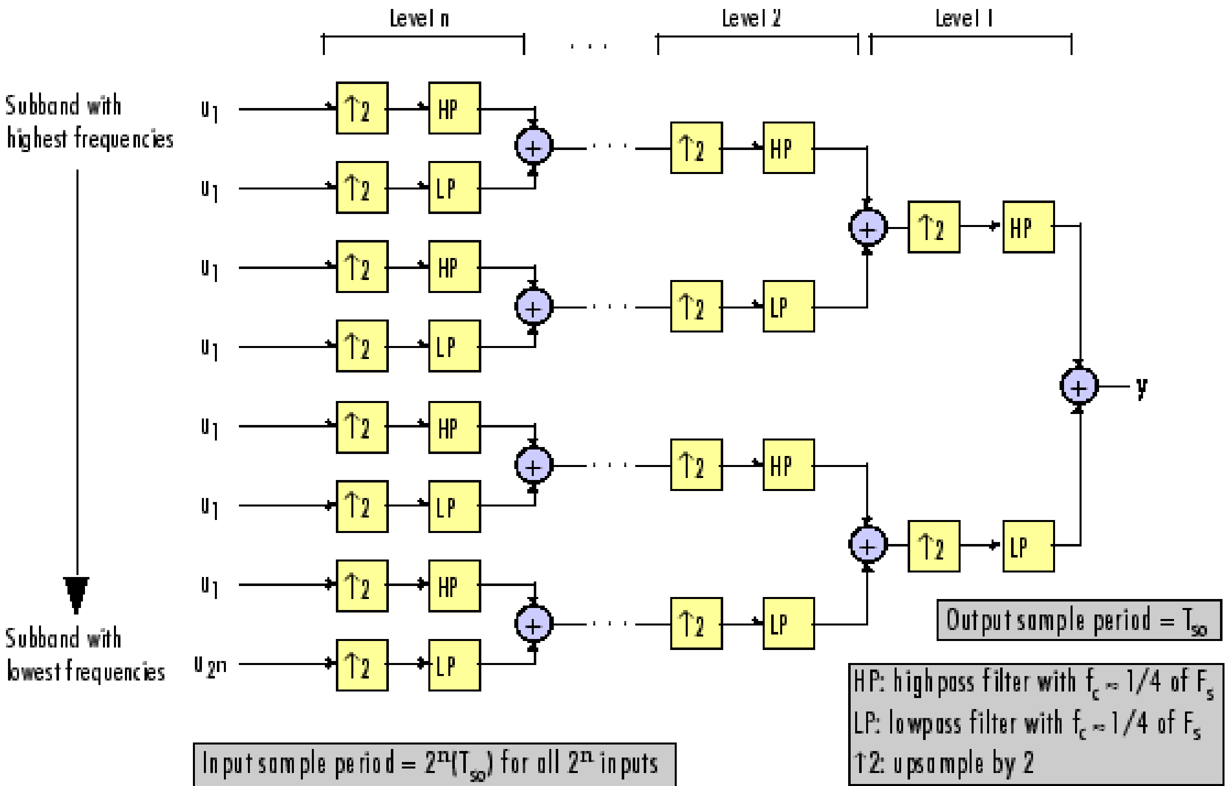
The unit takes in adjacent high-frequency and low-frequency subbands, and reconstructs them into a wide-band signal. Compared to each subband input, the output has twice the bandwidth and twice the sample rate.

Note The following figures illustrate the *concept* of a filter bank, but *not* how the block implements a filter bank; the block uses a more efficient polyphase implementation.



n-Level Asymmetric Dyadic Synthesis Filter Bank

Use the above figure and the following figure to compare the two tree structures of the dyadic synthesis filter bank. Note that in the asymmetric structure, the low-frequency subband input to each level is the output of the previous level, while the high-frequency subband input to each level is an input to the filter bank. In the symmetric structure, both the low- and high-frequency subband inputs to each level are outputs from the previous level.



n-Level Symmetric Dyadic Synthesis Filter Bank

The following table summarizes the key characteristics of symmetric and asymmetric dyadic synthesis filter banks.

Notable Characteristics of Asymmetric and Symmetric Dyadic Synthesis Filter Banks

Characteristic	N-Level Symmetric	N-Level Asymmetric
Input Paths Through the Filter Bank	Both the high-frequency and low-frequency input subbands to each level (except the first) are the outputs of the previous level. The inputs to the first level are the inputs to the filter bank.	The low-frequency subband input to each level (except the first) is the output of the previous level. The low-frequency subband input to the first level, and the high-frequency subband input to each level, are inputs to the filter bank.
Number of Input Subbands	2^n	n+1
Bandwidth and Number of Samples in Input Subbands	All inputs subbands have bandwidth BW / 2^n and N / 2^n samples, where the output has bandwidth BW and N samples.	For an output with bandwidth BW and N samples, the *k*th input subband has the following bandwidth and number of samples. $BW_k = \begin{cases} BW/2^k & (1 \le k \le n) \\ BW/2^n & (k = n+1) \end{cases}$ $N_k = \begin{cases} N/2^k & (1 \le k \le n) \\ N/2^n & (k = n+1) \end{cases}$
Input Sample Periods	All input subbands have a sample period of $2^n(T_{so})$, where the output sample period is T_{so}.	Sample period of *k*th input subband $= \begin{cases} 2^k(T_{so}) & (1 \le k \le n) \\ 2^n(T_{so}) & (k = n+1) \end{cases}$ where the output sample period is T_{so}.

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 *analysis* filter bank is exactly canceled in the reconstruction of the signal in the dyadic *synthesis* filter bank.	

For more information, see Dyadic Synthesis Filter Bank.

Multirate Filtering Examples

Signal Processing Blockset 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 Signal Processing Blockset demos, type `demo blockset signal` 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	`dspaudiosrc`
Sigma-Delta A/D Converter	Illustrates analog-to-digital conversion using a sigma-delta algorithm implementation	`dspsdadc`
Wavelet Reconstruction and Noise Reduction	Uses the Dyadic Analysis Filter Bank and Dyadic Synthesis Filter Bank blocks to show both the perfect reconstruction property of wavelets and an application for noise reduction	`dspwavelet`

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	`doc_mrf_nbpf`
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	`doc_mrf_nhpf`
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	`doc_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	`doc_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	`doc_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	`doc_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	`doc_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	`doc_mrf_nlp`

Multirate Filtering Example Models	Description	Command for Opening Example Models in MATLAB
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	`doc_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	`doc_mrf_wlp`

Transforms

The Signal Processing Blockset Transforms library provides blocks for a number of transforms that are of particular importance in signal processing applications.

- "Transforming Time-Domain Data into the Frequency Domain" on page 4-2
- "Transforming Frequency-Domain Data into the Time Domain" on page 4-7
- "Linear and Bit-Reversed Output Order" on page 4-12
- "Calculating the Channel Latencies Required for Wavelet Reconstruction" on page 4-14

Transforming Time-Domain Data into the Frequency Domain

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.

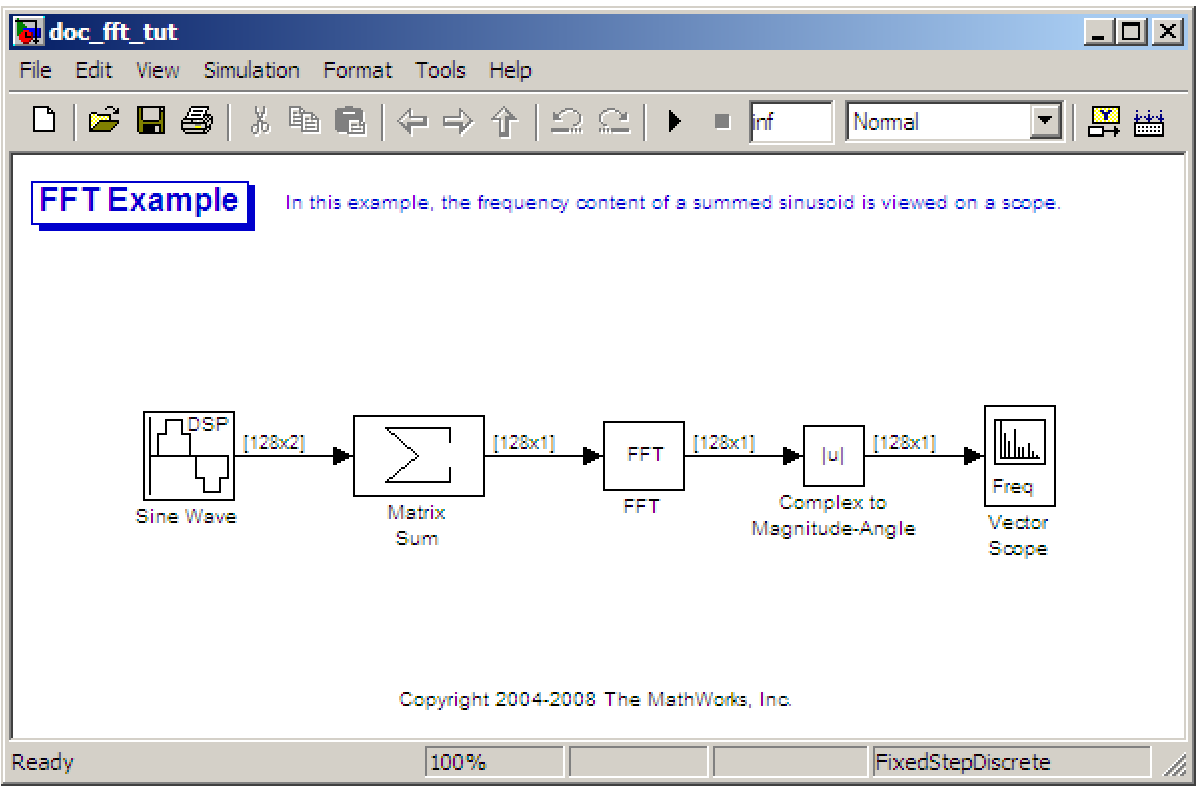
In this example, you use the Sine Wave block to generate two frame-based sinusoids, one at 15 Hz and the other at 40 Hz. You sum the sinusoids point-by-point to generate the compound sinusoid

$$u = \sin(30\pi t) + \sin(80\pi t)$$

Then, you transform this sinusoid into the frequency domain using an FFT block:

1. At the MATLAB command prompt, type `doc_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, frame-based sinusoidal signals with identical amplitudes, phases, and sample times. One sinusoid oscillates at 15 Hz and the other at 40 Hz.

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

5. Double-click the Matrix Sum block. The **Block Parameters: Matrix Sum** dialog box opens.

6. Set the **Sum over** parameter to `Specified dimension` and the **Dimension** parameter to `2`. Click **OK** to save your changes.

   Because each column represents a different signal, you need to sum along the individual rows in order to add the values of the sinusoids at each time step.

7. Double-click the Complex to Magnitude-Angle block. The **Block Parameters: Complex to Magnitude-Angle** dialog box opens.

8. Set the **Output** parameter to `Magnitude`, and then click **OK**.

   This block takes the complex output of the FFT block and converts this output to magnitude.

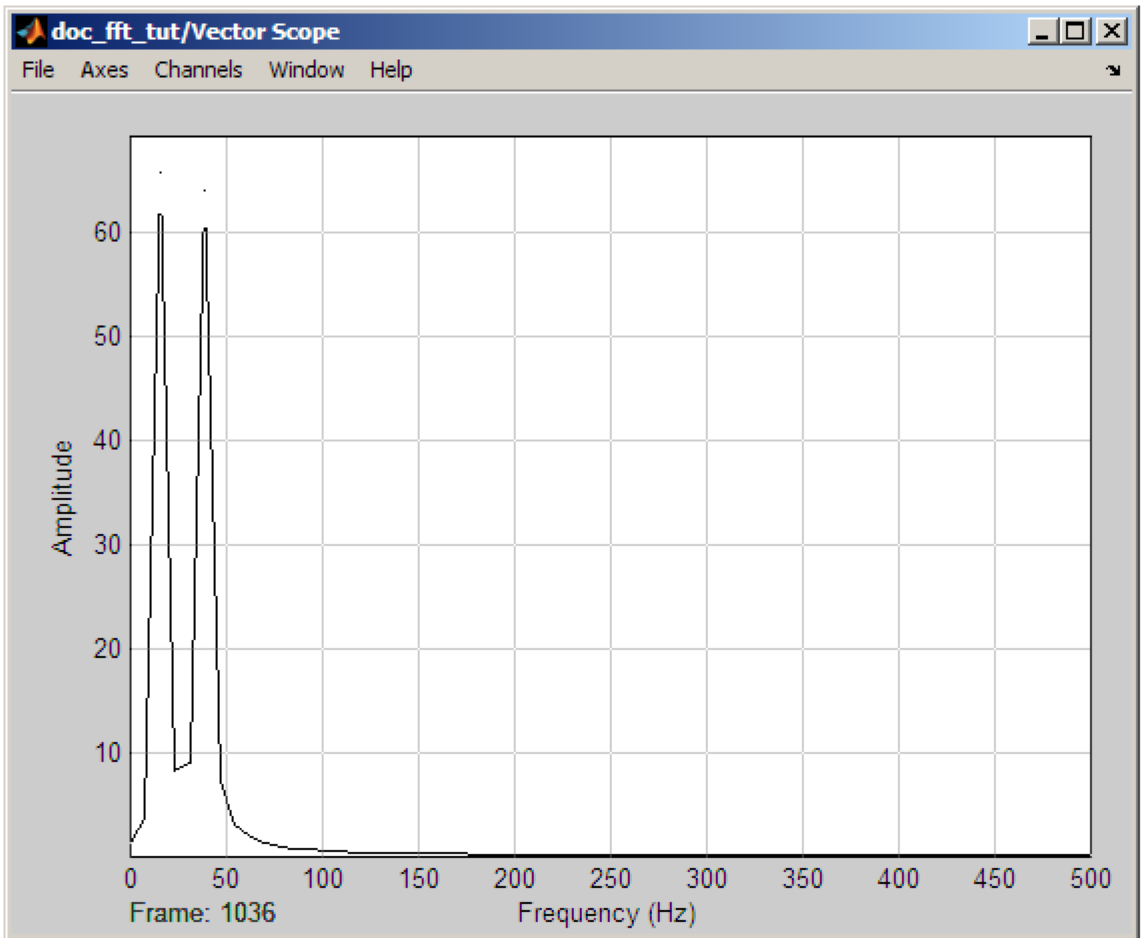
9. Double-click the Vector Scope block.

10. Set the block parameters as follows, and then click **OK**:

    - Click the **Scope Properties** tab.
    - **Input domain** = `Frequency`
    - Click the **Axis Properties** tab.
    - **Frequency units** = `Hertz` (This corresponds to the units of the input signals.)
    - **Frequency range** = `[0...Fs/2]`
    - Select the **Inherit sample time from input** check box.
    - **Amplitude scaling** = `Magnitude`

11 Run the model.

The scope shows the two peaks at 15 and 40 Hz, as expected.



You have now transformed two, frame-based 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 "Power Spectrum Estimation" on page 6-6 for more information about these blocks.

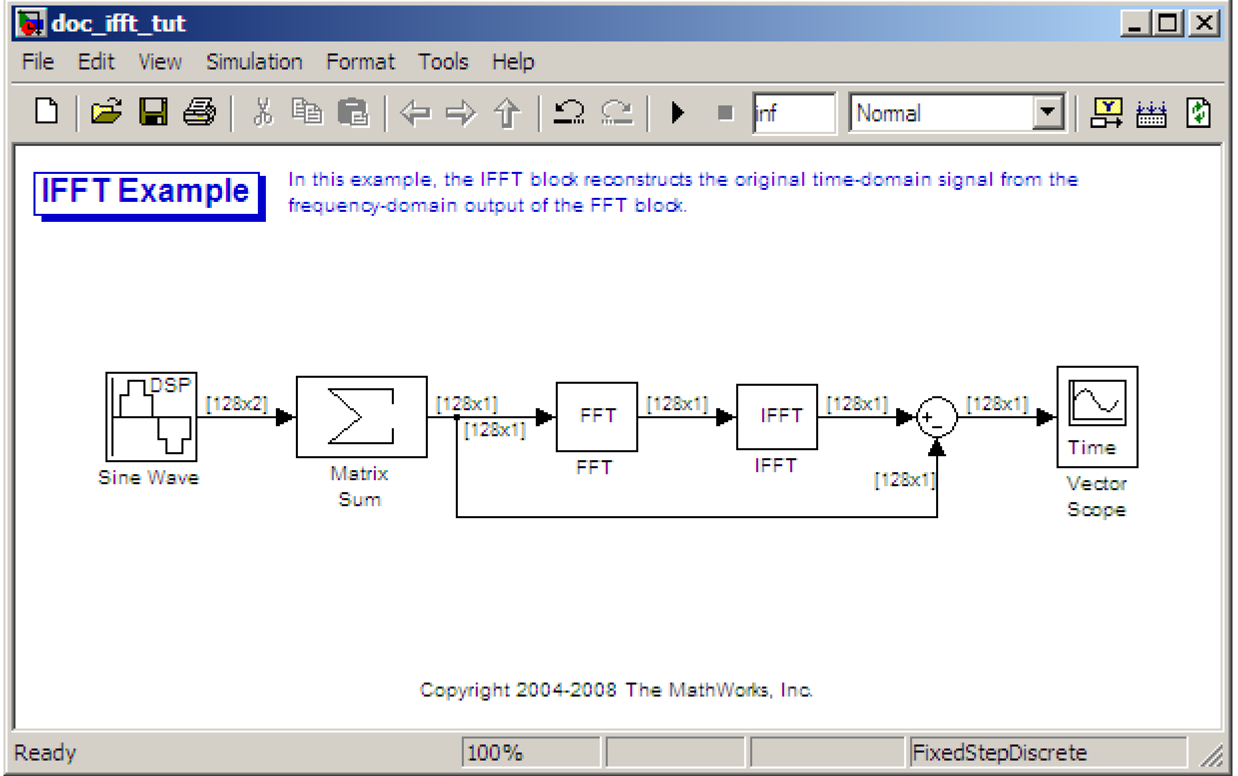
Transforming Frequency-Domain Data into the Time Domain

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 frame-based 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 `doc_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, frame-based 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.

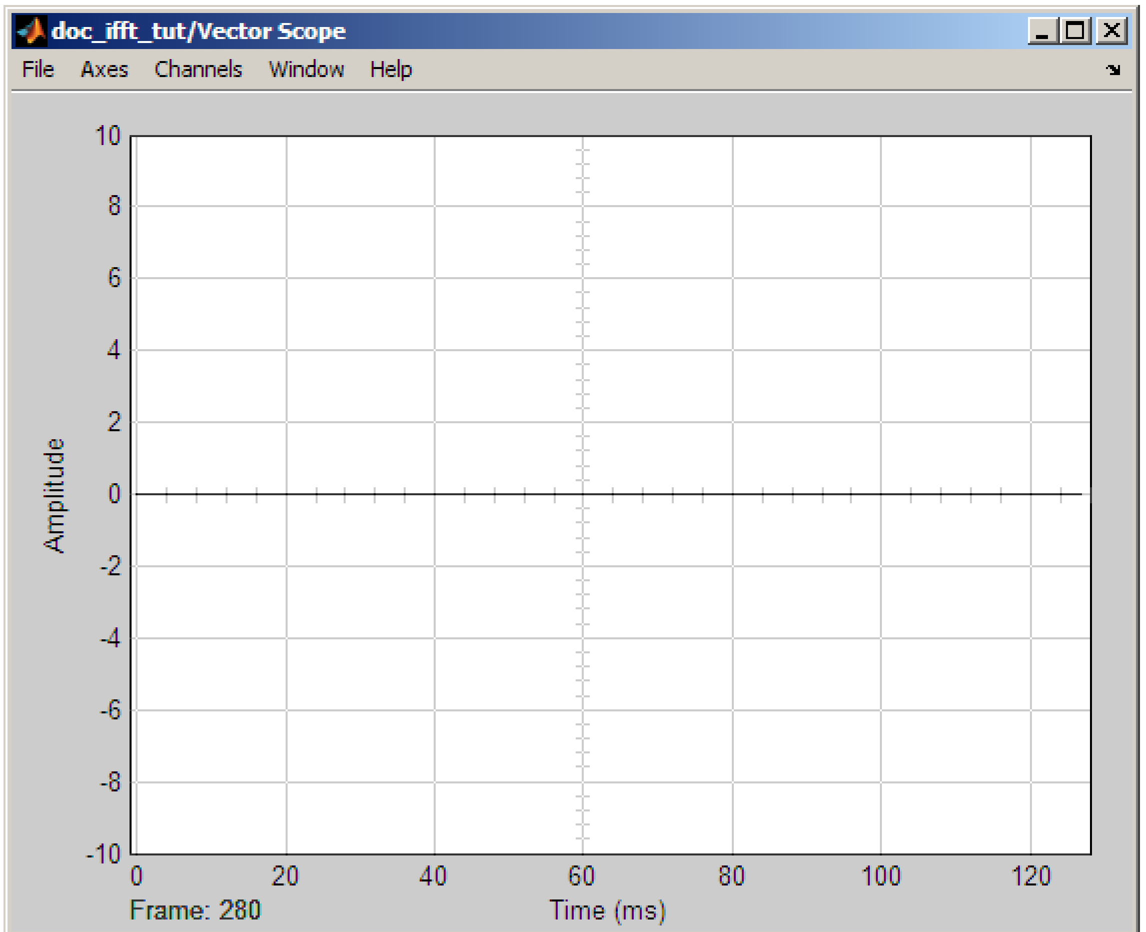
   Note that the Sum block subtracts the original signal from the output of the IFFT block, which is the estimation of the original signal.

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 4-12
"Finding the Bit-Reversed Order of Your Frequency Indices" on page 4-12

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.

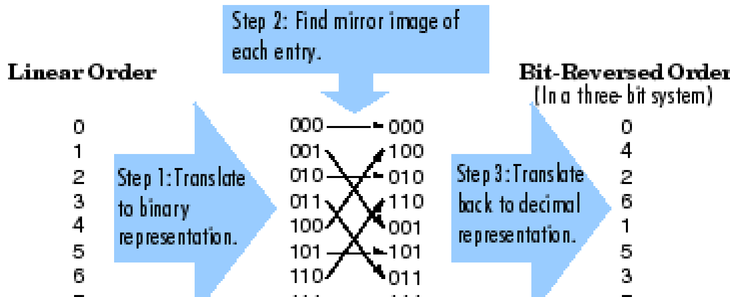
The input to the IFFT block can be in linear or bit-reversed order. Therefore, you do not have to alter the ordering of your data before transforming it back into the time domain. However, the IFFT block may run more quickly when the input is provided in bit-reversed order.

Finding the Bit-Reversed Order of Your Frequency Indices

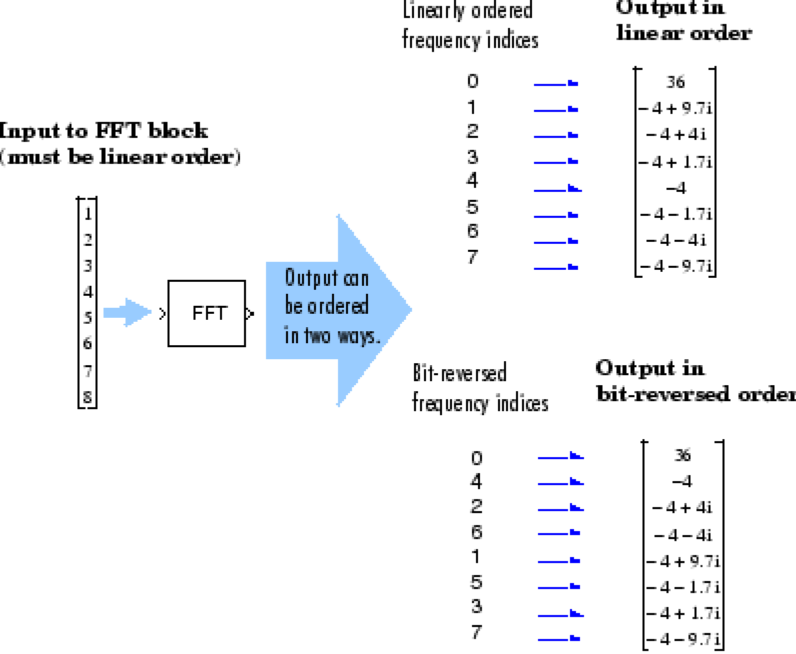
Two numbers are bit-reversed values of each other when the binary representation of one is the mirror image of the binary representation of the other. For example, in a three-bit system, one and four are bit-reversed values of each other, since the three-bit binary representation of one, 001, is the 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.



Calculating the Channel Latencies Required for Wavelet Reconstruction

In this section...

"Analyzing Your Model" on page 4-14

"Calculating the Group Delay of Your Filters" on page 4-16

"Reconstructing the Filter Bank System" on page 4-18

"Equalizing the Delay on Each Filter Path" on page 4-18

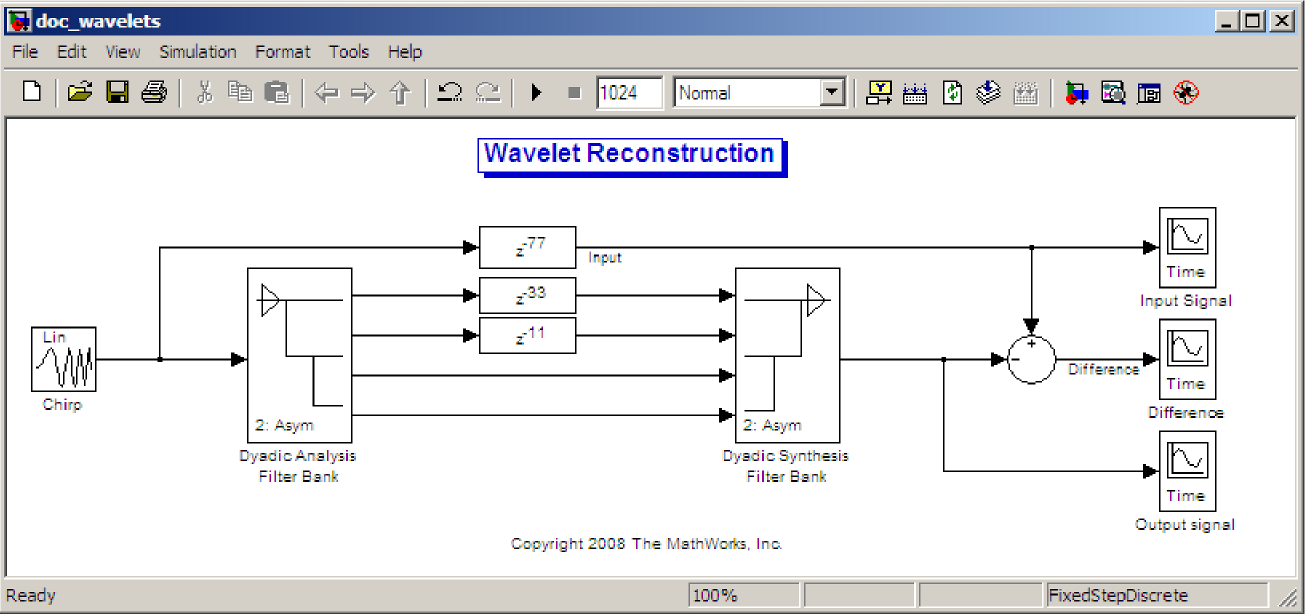
"Updating and Running the Model" on page 4-21

"References" on page 4-22

Analyzing Your Model

The following sections guide you through the process of calculating the channel latencies required for perfect wavelet reconstruction. This example uses the `doc_wavelets` model, but you can apply the process to perform perfect wavelet reconstruction in any model. To open the example model, type doc_wavelets at the MATLAB command line.

Note You must have a Wavelet Toolbox™ product license to run the `doc_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 `doc_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.

Calculating 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:

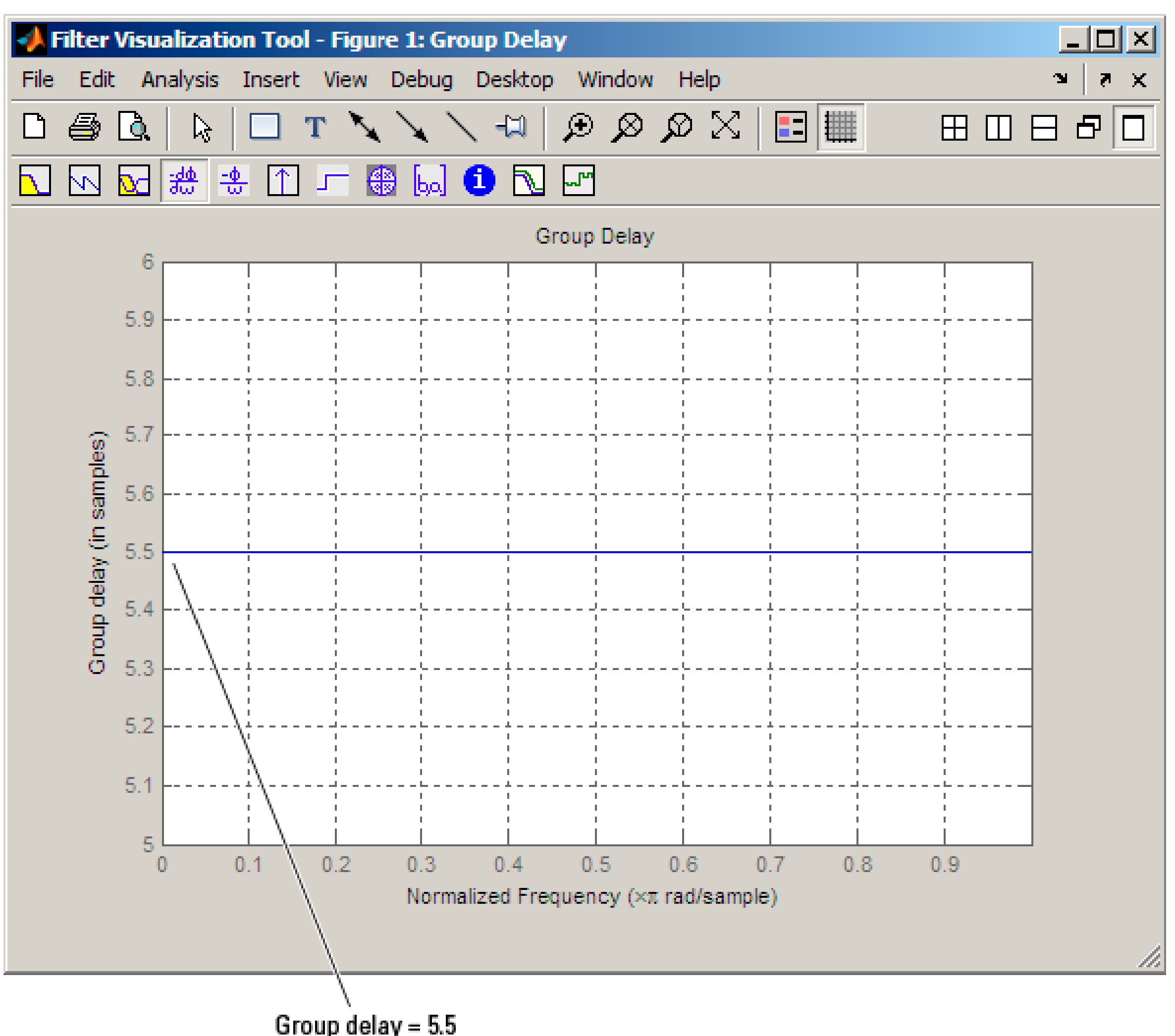
```
[Lo_D, Hi_D, Lo_R, Hi_R] = wfilters('bior3.5')
```

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.



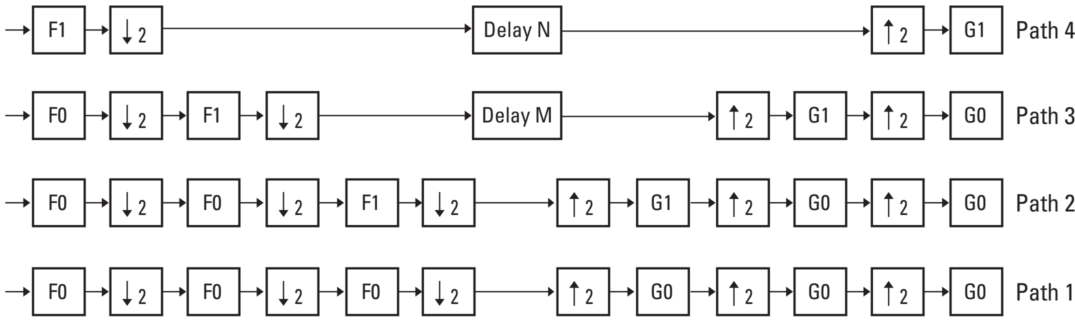
Note Repeat this procedure to analyze the group delay of each of the filters in your model. This section does not show the results for each filter in the `doc_wavelets` model because all wavelet filters in this particular example have the same group delay.

Reconstructing 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 Signal Processing Blockset User's Guide:

- "Dyadic Analysis Filter Banks" on page 3-66
- "Dyadic Synthesis Filter Banks" on page 3-70

Because the filter blocks in the `doc_wavelets` model use biorthogonal filters with three levels and an asymmetric tree structure, the filter bank system appears as shown in the following figure.



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

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.

Equalizing 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

→ z^{-2} → ↓2 → Equivalent to → ↓2 → z^{-1} →

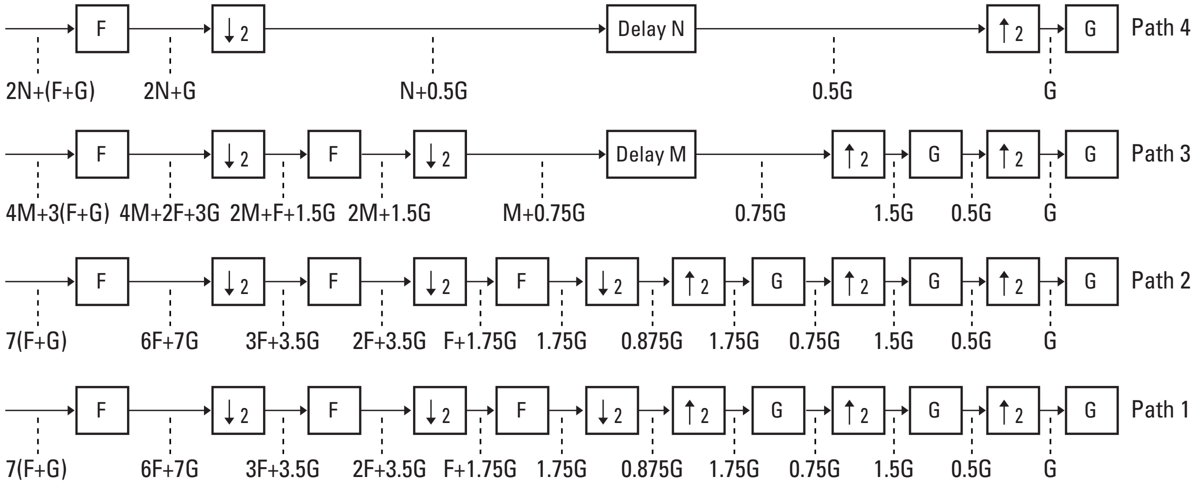
Second Noble Identity

→ ↑2 → z^{-2} → Equivalent to → z^{-1} → ↑2 →

You can apply the Noble identities by summing the delay on each signal path from right to left. The first Noble identity indicates that moving a delay of 1 before a downsample of 2 is equivalent to multiplying that delay value by 2. Similarly, the second Noble identity indicates that moving a delay of 2 before an upsample of 2 is equivalent to dividing that delay value by 2.

The `fvtool` analysis in step 1 found that both the low- and high-pass filters of the analysis filter bank have the same group delay ($F_0 = F_1 = 5.5$). Thus, you can use F to represent the group delay of the analysis filter bank. Similarly, the group delay of the low- and high-pass filters of the synthesis filter bank is the same ($G_0=G_1=5.5$), so you can use G to represent the group delay of the synthesis filter bank.

The following figure shows the filter bank system with the intermediate delay sums displayed below each path.



F = Delay due to Dyadic Analysis Filter Bank
G = Delay due to Dyadic Synthesis Filter Bank

You can see from the previous figure that the signal delays on paths 1 and 2 are identical: $7(F{+}G)$. Because each path of the filter bank system has identical delay, you can equate the delay equations for paths 3 and 4 with the delay equation for paths 1 and 2. After constructing these equations, you can solve for M and N, respectively:

$$\text{Path 3} = \text{Path 1} \Rightarrow 4M + 3(F+G) = 7(F+G)$$
$$\Rightarrow M = F + G$$

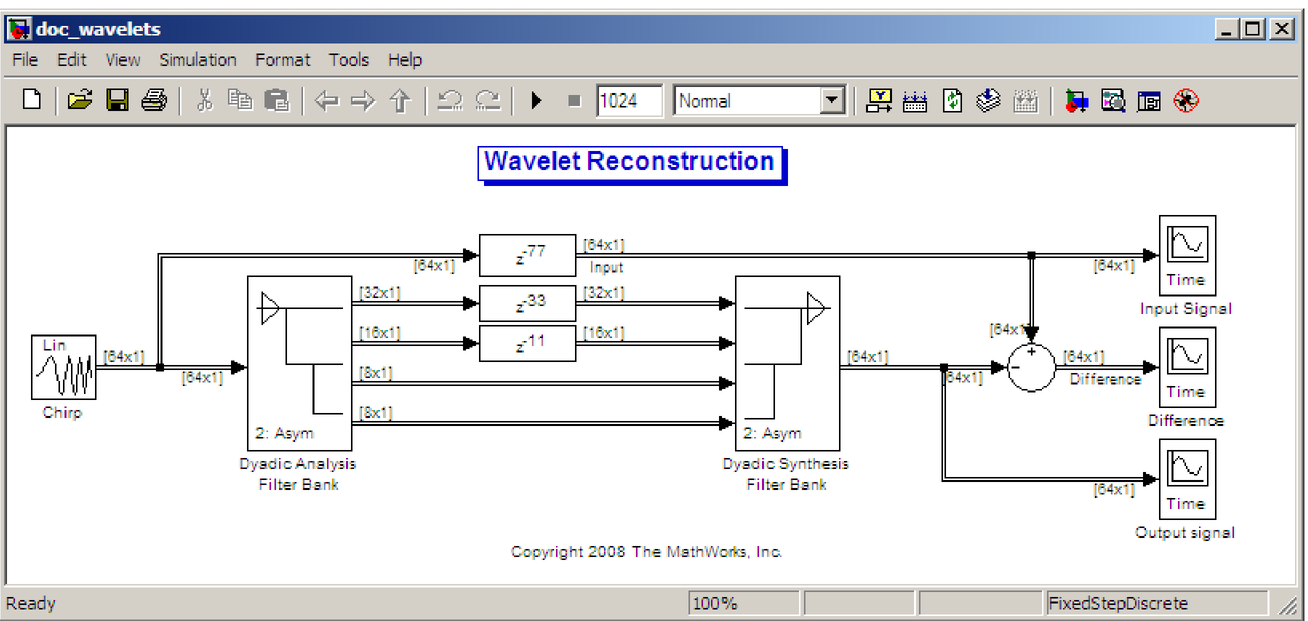
$$\text{Path 4} = \text{Path 1} \Rightarrow 2N + (F+G) = 7(F+G)$$
$$\Rightarrow N = 3(F+G)$$

The `fvtool` analysis in step 1 found the group delay of each biorthogonal wavelet filter in this model to be 5.5 samples. Therefore, $F = 5.5$ and $G = 5.5$. By inserting these values into the two previous equations, you get $M = 11$ and $N = 33$. Because the total delay on each filter path must be the same, you can find the overall delay of the filter bank system by inserting $F = 5.5$

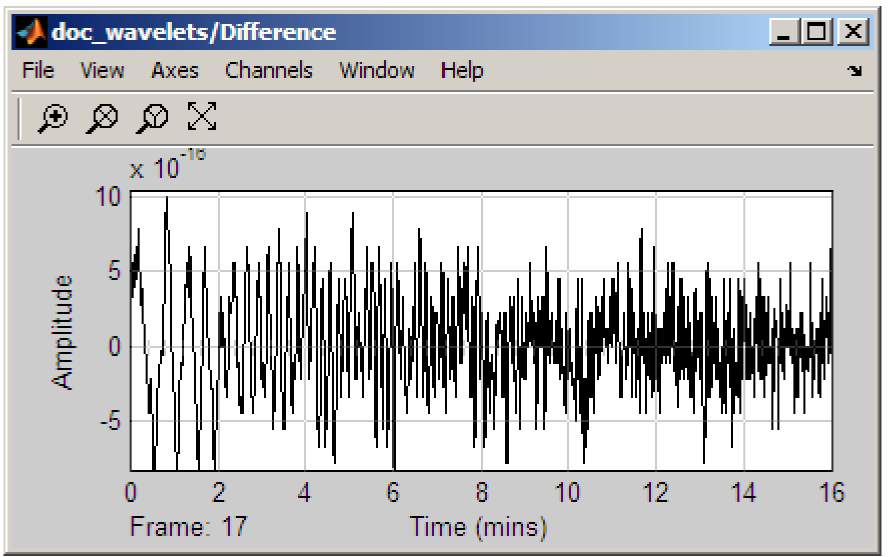
and $G = 5.5$ into the delay equation for any of the four filter paths. Inserting the values of F and G into $7(F+G)$ yields an overall delay of 77 samples for the filter bank system of the `doc_wavelets` model.

Updating and Running the Model

Now that you know the latencies required for perfect wavelet reconstruction, you can incorporate those delay values into the model. The `doc_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.

Quantizers

This chapter shows you how to design and use scalar and vector quantizer blocks. You create several scalar quantizer blocks and use them to encode and decode signals in your model. Then, you use vector quantizer encoder and decoder blocks to quantize vectors of data.

- "Scalar Quantizers" on page 5-2
- "Vector Quantizers" on page 5-10

Scalar Quantizers

In this section...

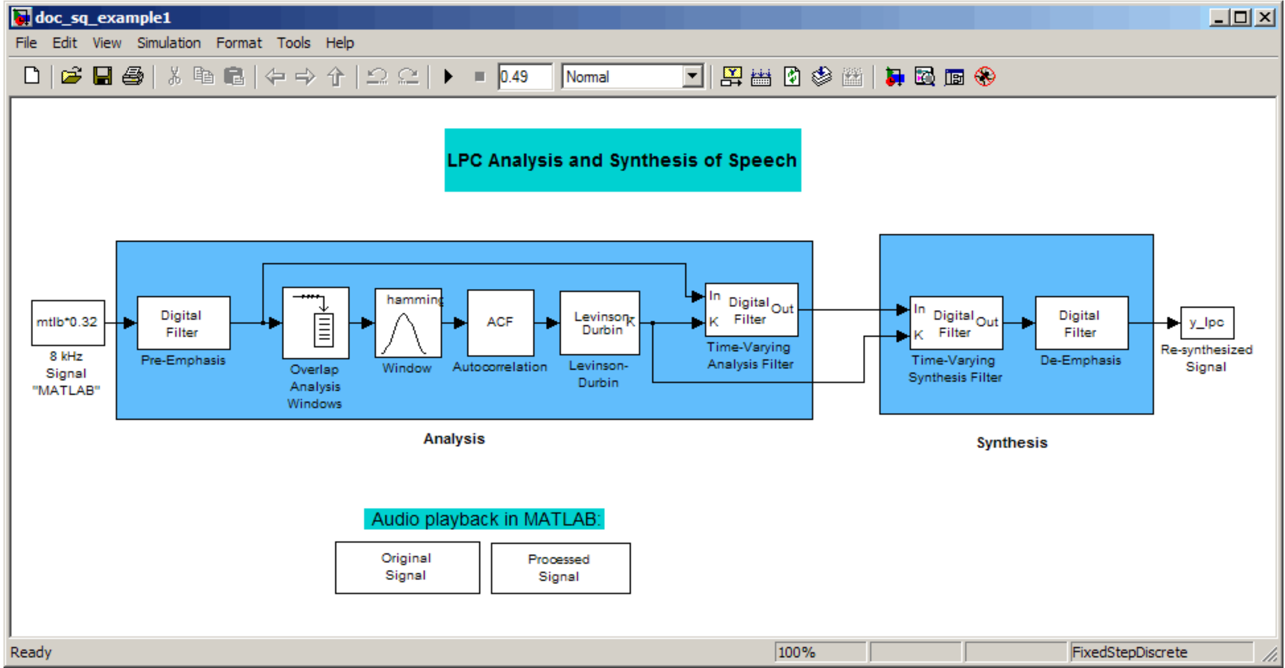
"Analysis and Synthesis of Speech" on page 5-2
"Identifying Your Residual Signal and Reflection Coefficients" on page 5-4
"Creating a Scalar Quantizer" on page 5-5

Analysis and Synthesis of Speech

You can use blocks from the Signal Processing Blockset 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 `doc_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.

Identifying Your Residual Signal and Reflection Coefficients

In the previous topic, "Analysis and Synthesis of Speech" on page 5-2, 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 `doc_sq_example1` at the MATLAB command line.

2. Save the model file as `doc_sq_example2` in your working folder.

3. From the Signal Processing Sinks library, click-and-drag two Signal To Workspace blocks into your model.

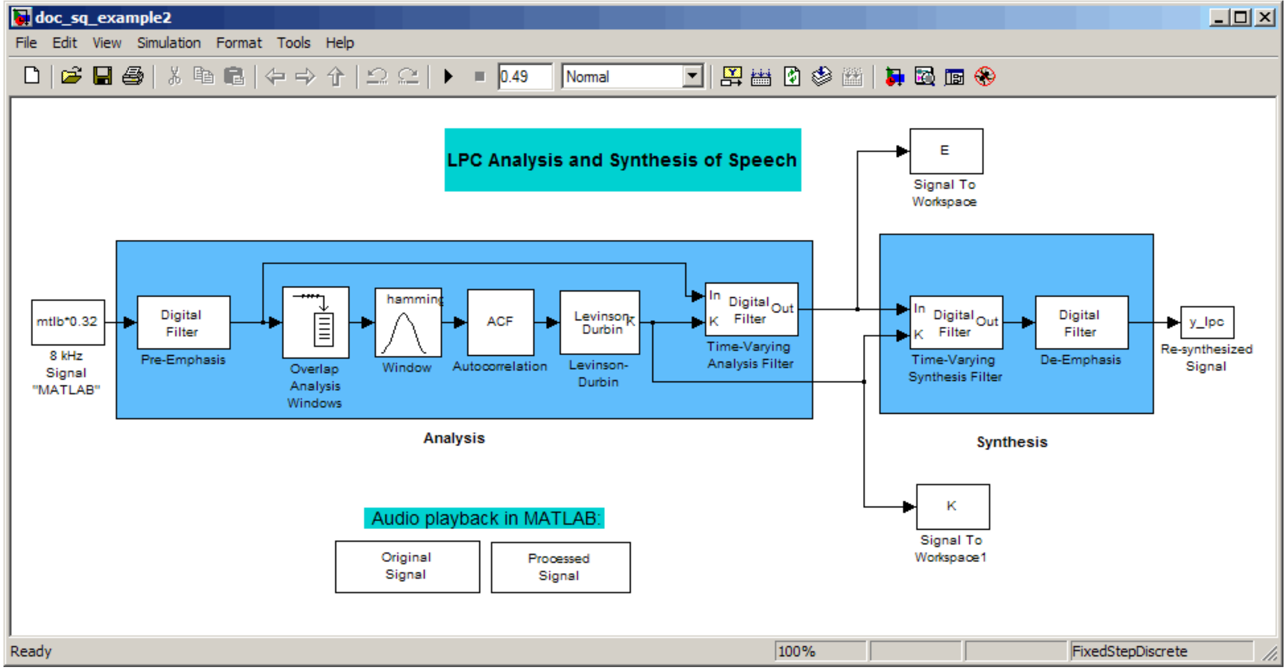
4. Connect the output of the Levinson-Durbin block to one of the Signal To Workspace blocks.

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.

Creating 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 "Identifying Your Residual Signal and Reflection Coefficients" on page 5-4 is not open on your desktop, you can open an

equivalent model by typing doc_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`.

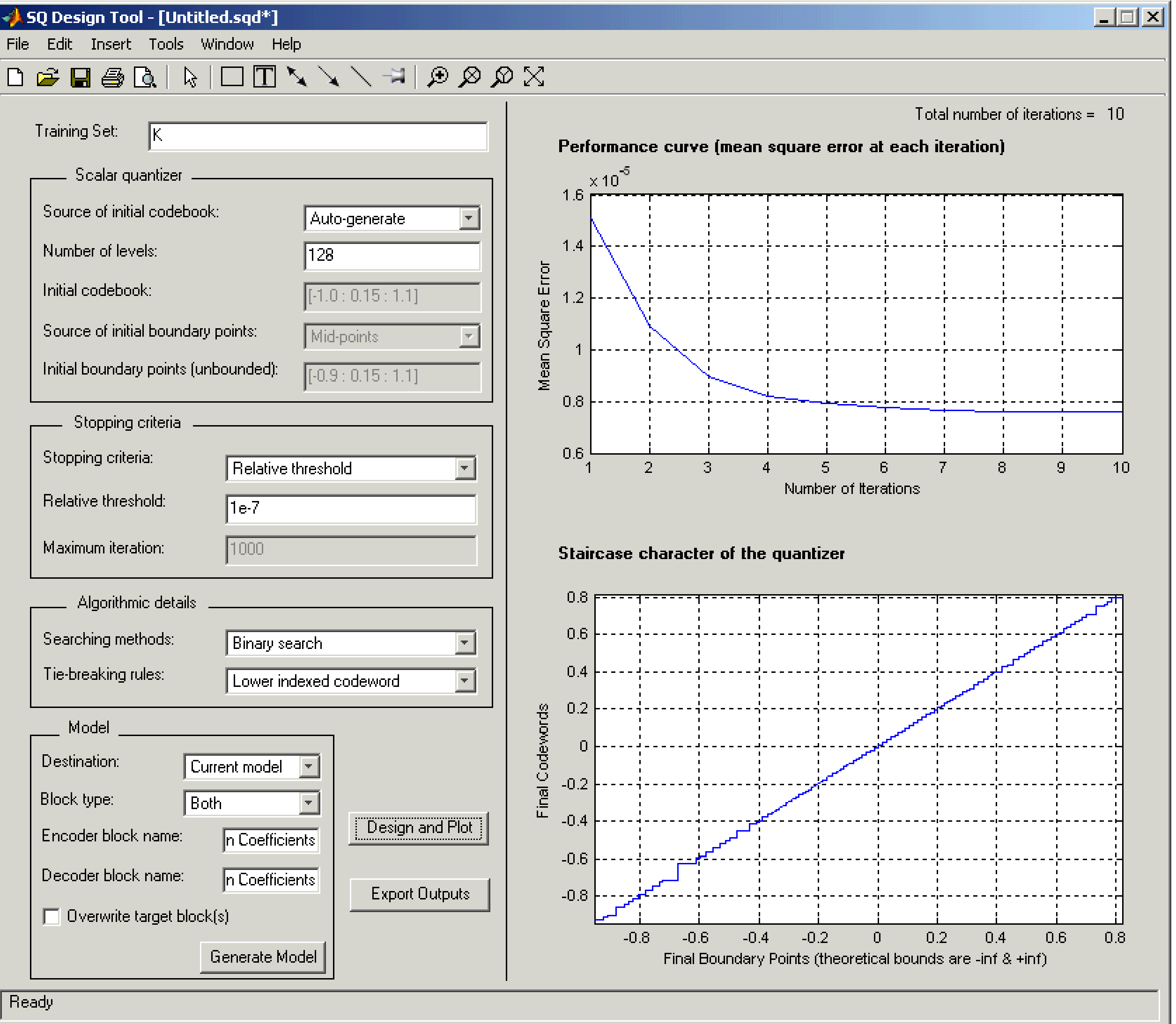
7 For the **Encoder block name** parameter, enter `SQ Encoder - Reflection Coefficients`.

8 For the **Decoder block name** parameter, enter `SQ Decoder - Reflection Coefficients`.

9 Make sure that your desired destination model, `doc_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`.

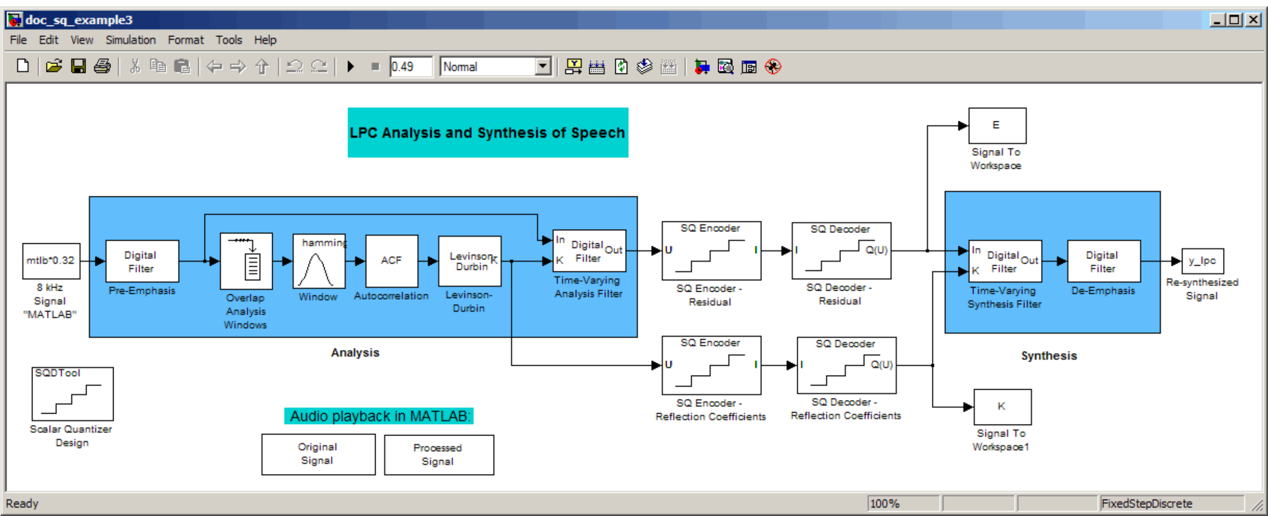
13 Repeat steps 5 to 11 for the variable *E*, which represents the residual signal you want to quantize. In steps 6 and 7, name your blocks SQ Encoder - Residual and SQ Decoder - Residual.

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 `doc_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 5-10.

Vector Quantizers

In this section...

"Building Your Vector Quantizer Model" on page 5-10

"Configuring and Running Your Model" on page 5-11

Building 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:

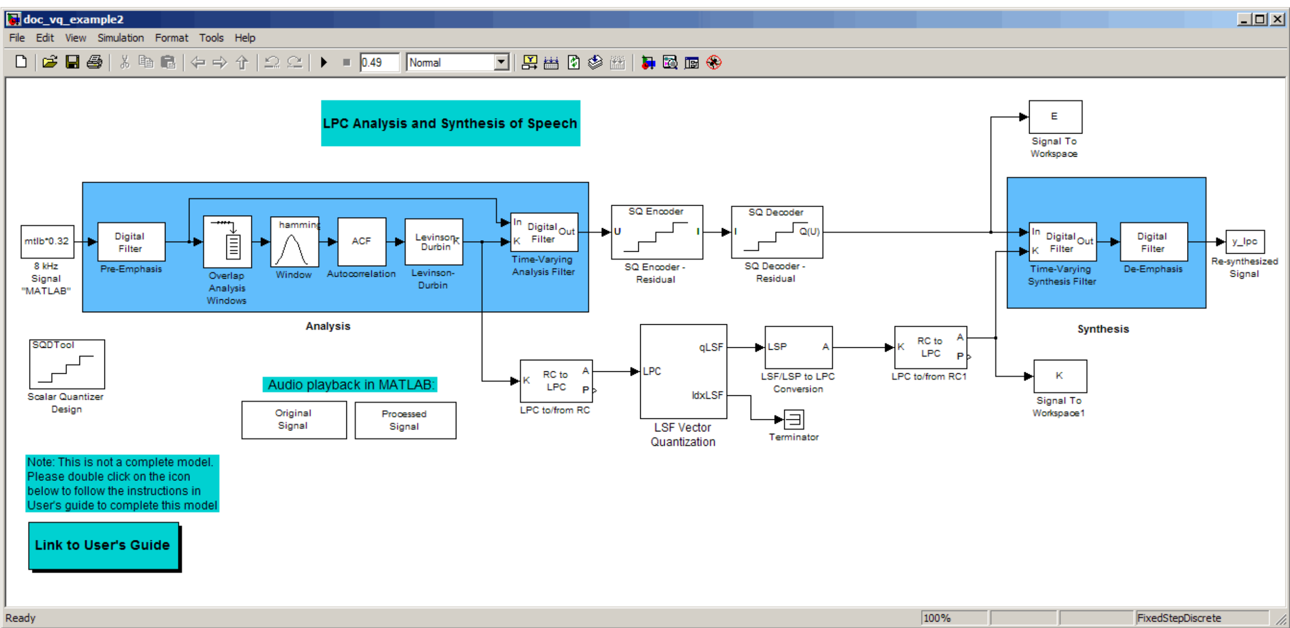
1. Open a model similar to the one you created in "Creating a Scalar Quantizer" on page 5-5 by typing doc_vq_example1 at the MATLAB command prompt. The example model `doc_vq_example1` adds a new LSF Vector Quantization subsystem to the `doc_sq_example3` model. This subsystem is preconfigured to work 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 Signal Processing Blockset 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.

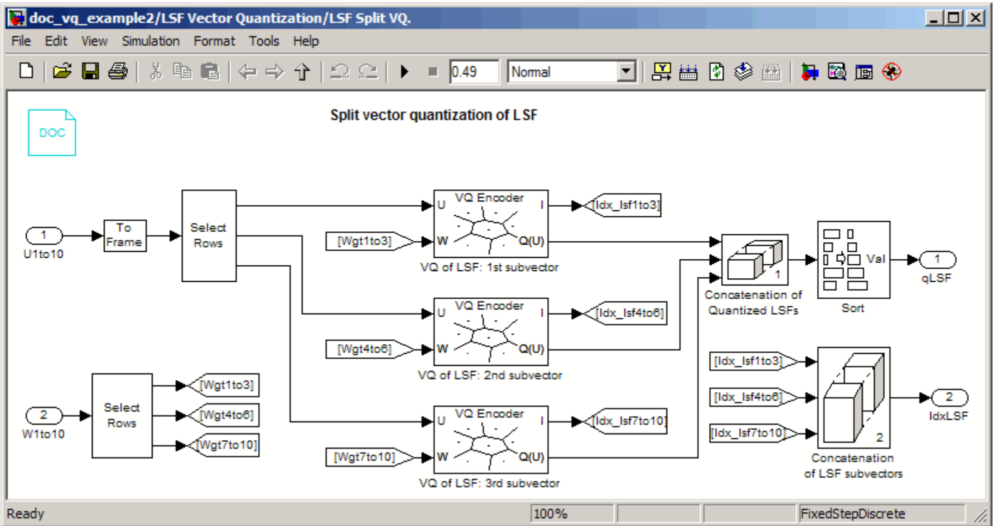
Configuring and Running Your Model

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 “Building Your Vector Quantizer Model” on page 5-10 is not open on your desktop, you can open an equivalent model by typing doc_vq_example2 at the MATLAB command prompt.

2 Double-click the LSF Vector Quantization subsystem, and then double-click the LSF Split VQ subsystem.

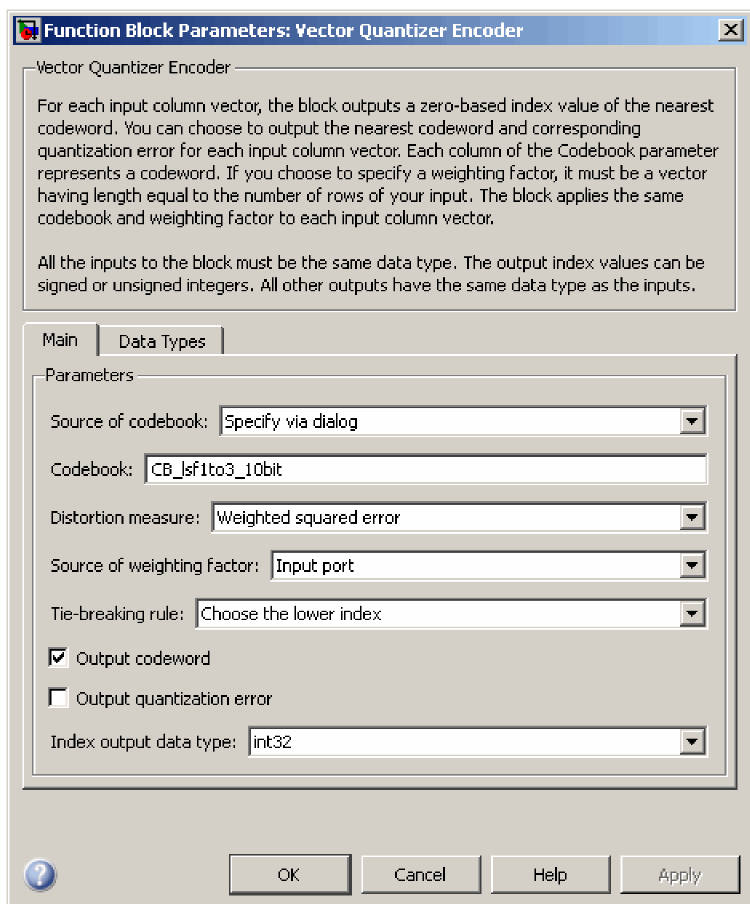
The subsystem opens, and you see the three Vector Quantizer Encoder blocks used to implement the split vector quantization method.



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**.

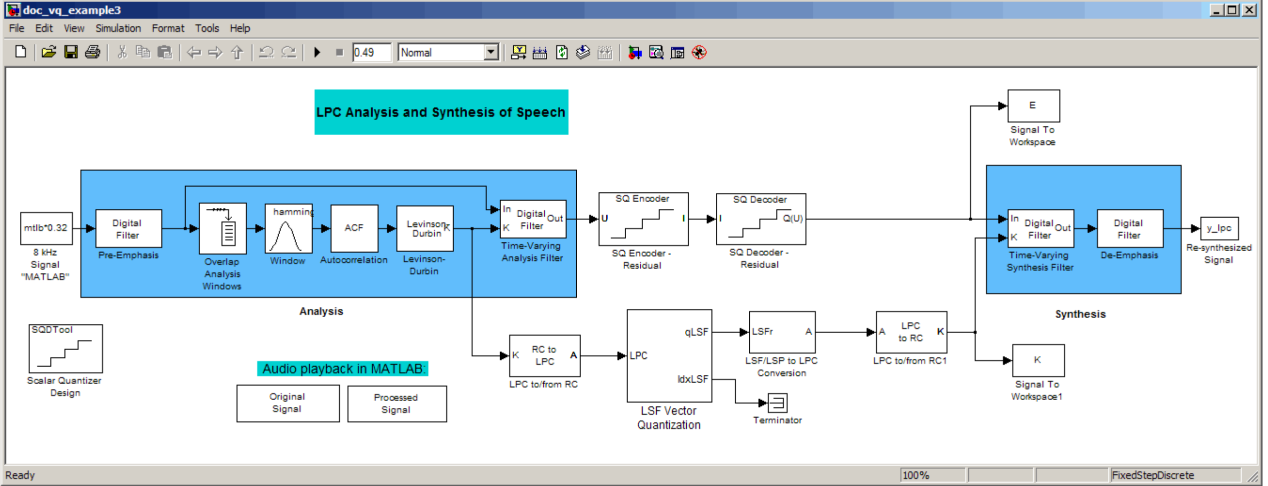
6. Double-click the LSF/LSP to LPC Conversion block and set the **Input** parameter to `LSF in range (0 to pi)`. Click **OK**.

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. At the MATLAB command prompt, type `load lpcvocoder`.

   The codebook values for your vector quantizer are loaded into memory. You have now configured the parameters of your vector quantizer model and are ready to quantize your reflection coefficients.

9. Run your model.



10 Double-click the Original Signal and Processed Signal blocks to listen to both the original and the processed signal.

There is no perceptible difference between the two. Quantizing your reflection coefficients using a split vector quantization method produced good quality speech without much distortion.

You have now used the split vector quantization method to quantize your reflection coefficients. The vector quantizers in the LSF Vector Quantization subsystem use 30 bits to quantize a frame containing 80 reflection coefficients. The bit rate of a quantization system is calculated as (bits per frame)*(frame rate).

In this example, the bit rate is [(80 residual samples/frame)*(7 bits/sample) + (30 bits/frame)]*(100 frames/second), or 59 kbits per second. This is less than 64.4 kbits per second, the bit rate of the scalar quantization system. However, the quality of the speech signal did not degrade. If you want to further reduce the bit rate of your system, you can use the vector quantization method to quantize the residual signal.

6

Statistics, Estimation, and Linear Algebra

This chapter describes several standard operations involved in simulating signal processing models.

- "Statistics" on page 6-2
- "Power Spectrum Estimation" on page 6-6
- "Linear Algebra" on page 6-7

Statistics

In this section...
"Statistics Blocks" on page 6-2
"Basic Operations" on page 6-3
"Running Operations" on page 6-4

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
```

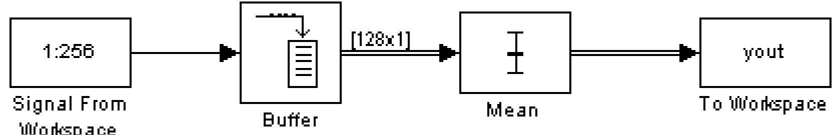
which computes the maximum of each column in input `u`.

The next section is an example of a basic statistical operation.

Example: Sliding Windows

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 model below.



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 (the same as the Delay Line block), so that a new statistic is computed for every new signal sample.

To build the model, make the following settings:

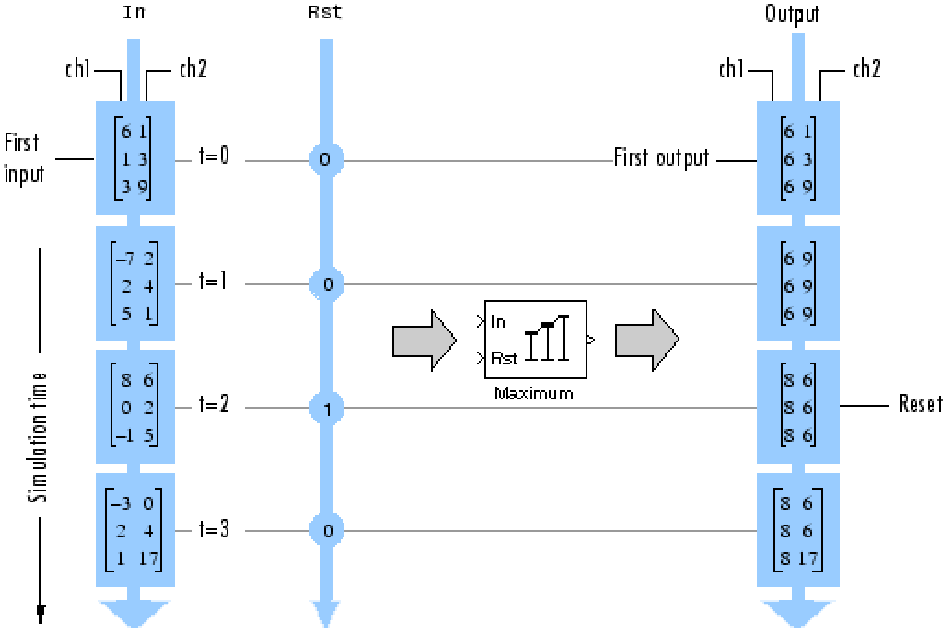
- In the Signal From Workspace block, set:
  - **Signal** = `1:256`
  - **Sample time** = `0.1`
  - **Samples per frame** = `1`
- In the Buffer block, set:
  - **Output buffer size (per channel)** = `128`
  - **Buffer overlap** = `127`

Running Operations

A *running operation* is one that processes successive sample-based or frame-based inputs, and computes a result that reflects both present and past 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 figure below illustrates how a Maximum block in running mode operates on a frame-based 3-by-2 (two-channel) matrix input, `u`. The running maximum is reset at t=2 by an impulse to the block's optional `Rst` port.



Power Spectrum Estimation

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.

Signal Processing Blockset software provides two demos that illustrate the spectral analysis blocks:

- A Comparison of Spectral Analysis Techniques (`dspsacomp`)
- Spectral Analysis: Short-Time FFT (`dspstfft`)

Linear Algebra

In this section...

"Linear Algebra Blocks" on page 6-7
"Linear System Solvers" on page 6-7
"Matrix Factorizations" on page 6-9
"Matrix Inverses" on page 6-11

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. See Chapter 1, "Working with Signals" for more information about matrix signals.

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.

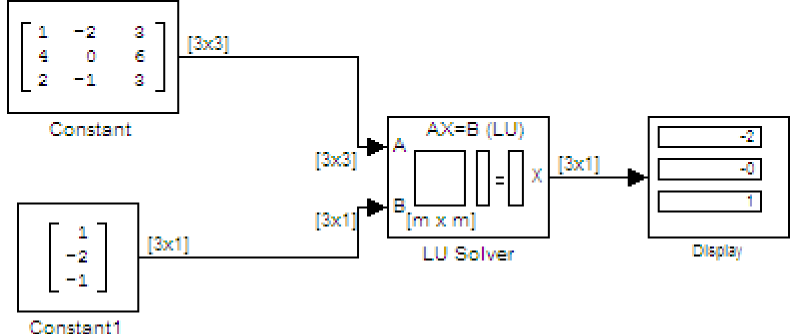
Example: LU Solver

In the model below, the LU Solver block solves the equation Ax = b, where

$$\mathbf{A} = \begin{bmatrix} 1 & -2 & 3 \\ 4 & 0 & 6 \\ 2 & -1 & 3 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} 1 \\ -2 \\ -1 \end{bmatrix}$$

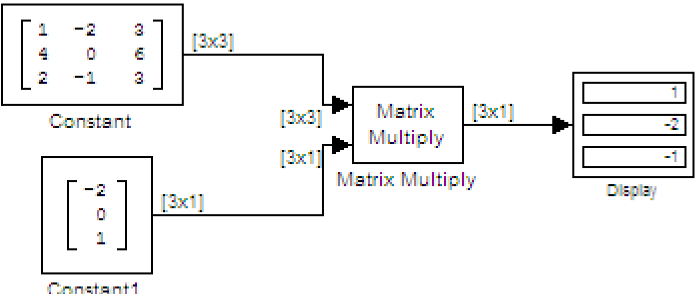
and finds x to be the vector `[-2 0 1]'`.

To build the model, set the following parameters:



- In the Constant block, set **Constant value** = `[1 -2 3;4 0 6;2 -1 3]`.
- In the Constant1 block, set **Constant value** = `[1 -2 -1]'`.
- In both Constant blocks, clear the **Interpret vector parameters as 1-D** check box.
- In both Constant blocks, set the **Sampling mode** to `Sample based`.
- In both Constant blocks, set the **Sample time** to `1`.

You can verify the solution by using the Matrix Multiply block to perform the multiplication Ax, as shown in the model below.



Matrix Factorizations

The Matrix Factorizations library provides the following blocks for factoring various kinds of matrices:

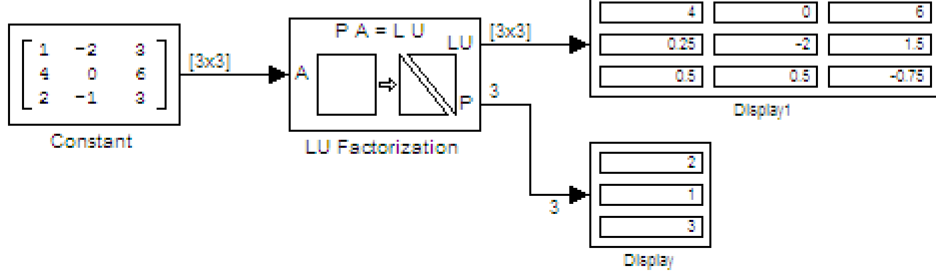
- Cholesky Factorization
- LDL Factorization
- LU Factorization
- QR Factorization
- Singular Value Decomposition

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.

Example: LU Factorization

In the model below, 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

$$\mathbf{A} = \begin{bmatrix} 1 & -2 & 3 \\ 4 & 0 & 6 \\ 2 & -1 & 3 \end{bmatrix}$$



To build the model, in the DSP Constant block, set the **Constant value** parameter to `[1 -2 3;4 0 6;2 -1 3]`, clear the **Interpret vector parameters as 1–D** check box, set the**Sampling mode** to `Sample based`, and set the **Sample time** to `1`.

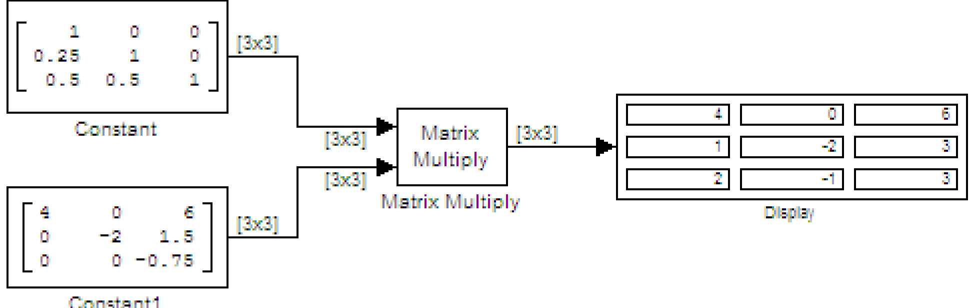
The lower output of the LU Factorization, `P`, is the permutation index vector, which indicates that the factored matrix A_p is generated from A by interchanging the first and second rows.

$$\mathbf{A}_p = \begin{bmatrix} 4 & 0 & 6 \\ 1 & -2 & 3 \\ 2 & -1 & 3 \end{bmatrix}$$

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 model below.



Matrix Inverses

The Matrix Inverses library provides the following blocks for inverting various kinds of matrices:

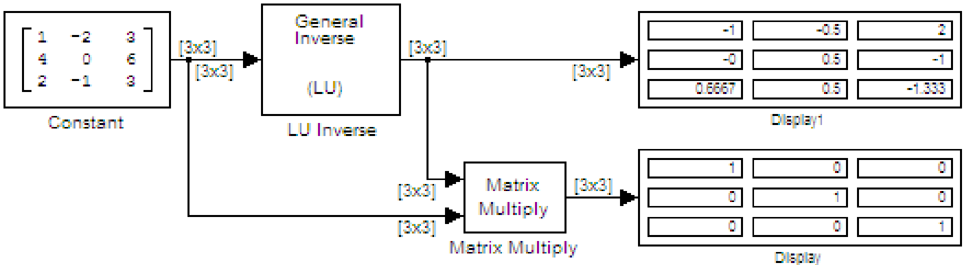
- Cholesky Inverse
- LDL Inverse
- LU Inverse
- Pseudoinverse

Example: LU Inverse

In the model below, the LU Inverse block computes the inverse of input matrix A, where

$$\mathbf{A} = \begin{bmatrix} 1 & -2 & 3 \\ 4 & 0 & 6 \\ 2 & -1 & 3 \end{bmatrix}$$

and then forms the product A^{-1}A, which yields the identity matrix of order 3, as expected.



To build the model, set the Constant block parameters as follows:

- **Constant value** = `[1 -2 3;4 0 6;2 -1 3]`
- **Interpret vector parameters as 1–D** = Clear this check box
- **Sampling mode** = `Sample based`
- **Sample time** = `1`

As shown above, the computed inverse is

$$\mathbf{A}^{-1} = \begin{bmatrix} -1 & -0.5 & 2 \\ 0 & 0.5 & -1 \\ 0.6667 & 0.5 & -1.333 \end{bmatrix}$$

Working with Fixed-Point Data

- "Fixed-Point Signal Processing Development" on page 7-2
- "Concepts and Terminology" on page 7-5
- "Arithmetic Operations" on page 7-11
- "Specifying Fixed-Point Attributes" on page 7-22
- "Fixed-Point Filtering" on page 7-45

Fixed-Point Signal Processing Development

In this section...

"Fixed-Point Features" on page 7-2

"Benefits of Fixed-Point Hardware" on page 7-2

"Benefits of Fixed-Point Design with Signal Processing Blockset Software" on page 7-3

"Fixed-Point Signal Processing Applications" on page 7-4

Note To take full advantage of fixed-point support in Signal Processing Blockset software, you must install Simulink® Fixed Point™ software.

Fixed-Point Features

Many of the blocks in Signal Processing Blockset software have fixed-point support, so you can design signal processing systems that use fixed-point arithmetic. Fixed-point support in Signal Processing Blockset 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 Real-Time Workshop C code-generation target
- Overflow handling and rounding methods
- C code generation for deployment on a fixed-point embedded processor, with Real-Time Workshop 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 Signal Processing Blockset Software

Simulating your fixed-point development choices before implementing them in hardware saves time and money. The built-in fixed-point operations provided by Signal Processing Blockset software save time in simulation and allow you to generate code automatically.

Signal Processing Blockset 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 Signal Processing Blockset software and Real-Time Workshop code generation software produces code ready for execution on a fixed-point processor. All the choices you make in simulation with Signal Processing Blockset software 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 in the *Simulink Fixed Point User's Guide*.

Fixed-Point Signal Processing Applications

Fixed-point support in Signal Processing Blockset software facilitates development of a wide variety of signal processing applications:

- Wireless and broadband communications
  - Cellular phones
  - Radio
  - Satellite communications
- Speech and audio processing
  - Speech processing
  - High-end audio processing
- Telephony
  - Speech coding
  - Dual tone multifrequency (DTMF)
  - Echo cancellation
- Hand-held and battery-operated consumer electronics
  - Digital recording devices
  - Personal digital assistants (PDAs)
- Computer peripherals
- Radar and sonar
- Medical electronics

Concepts and Terminology

In this section...

"Fixed-Point Data Types" on page 7-5

"Scaling" on page 7-6

"Precision and Range" on page 7-7

Note The "Glossary" defines much of the vocabulary used in these sections. For more information on these subjects, see the Simulink Fixed Point documentation.

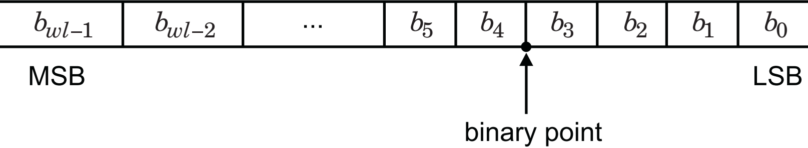
Fixed-Point Data Types

In digital hardware, numbers are stored in binary words. A binary word is a fixed-length sequence of bits (1's and 0's). How hardware components or software functions interpret this sequence of 1's and 0's is defined by the data type.

Binary numbers are represented as either fixed-point or floating-point data types. In this section, we discuss many terms and concepts relating to fixed-point numbers, data types, and mathematics.

A fixed-point data type is characterized by the word length in bits, the position of the binary point, and whether it is signed or unsigned. The position of the binary point is the means by which fixed-point values are scaled and interpreted.

For example, a binary representation of a generalized fixed-point number (either signed or unsigned) is shown below:



where

- b_i is the ith binary digit.
- wl is the word length in bits.
- b_{wl-1} is the location of the most significant, or highest, bit (MSB).
- b_0 is the location of the least significant, or lowest, bit (LSB).
- The binary point is shown four places to the left of the LSB. In this example, therefore, the number is said to have four fractional bits, or a fraction length of four.

Fixed-point data types can be either signed or unsigned. Signed binary fixed-point numbers are typically represented in one of three ways:

- Sign/magnitude
- One's complement
- Two's complement

Two's complement is the most common representation of signed fixed-point numbers and is used by Signal Processing Blockset software. See "Two's Complement" on page 7-12 for more information.

Scaling

Fixed-point numbers can be encoded according to the scheme

$$real\text{-}world\ value = (slope \times integer) + bias$$

where the slope can be expressed as

$$slope = slope\ adjustment \times 2^{exponent}$$

The integer is sometimes called the *stored integer*. This is the raw binary number, in which the binary point assumed to be at the far right of the word. In Signal Processing Blockset software, 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:

$$\textit{real-world value} = 2^{\textit{exponent}} \times \textit{integer}$$

or

$$\textit{real-world value} = 2^{-\textit{fraction length}} \times \textit{integer}$$

In Signal Processing Blockset 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.

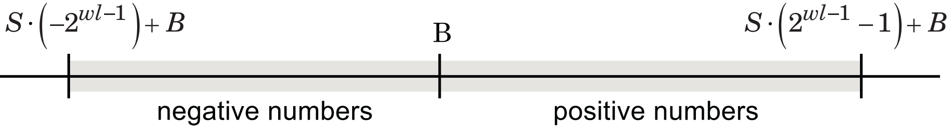
All Signal Processing Blockset blocks that support fixed-point data types support signals with binary-point only scaling. Many fixed-point Signal Processing Blockset blocks that do not perform arithmetic operations but merely rearrange data, such as Delay and Matrix Transpose, also support signals with [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

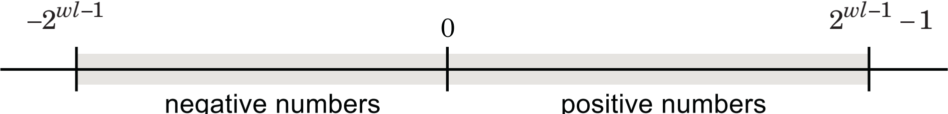
The range is the span of numbers that a fixed-point data type and scaling can represent. The range of representable numbers for a two's complement fixed-point number of word length wl, scaling S, and bias B is illustrated below:



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.

Signal Processing Blockset 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, Signal Processing Blockset 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 7-11 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, Signal Processing Blockset 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 blockset.
- `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 7-11

"Two's Complement" on page 7-12

"Addition and Subtraction" on page 7-13

"Multiplication" on page 7-14

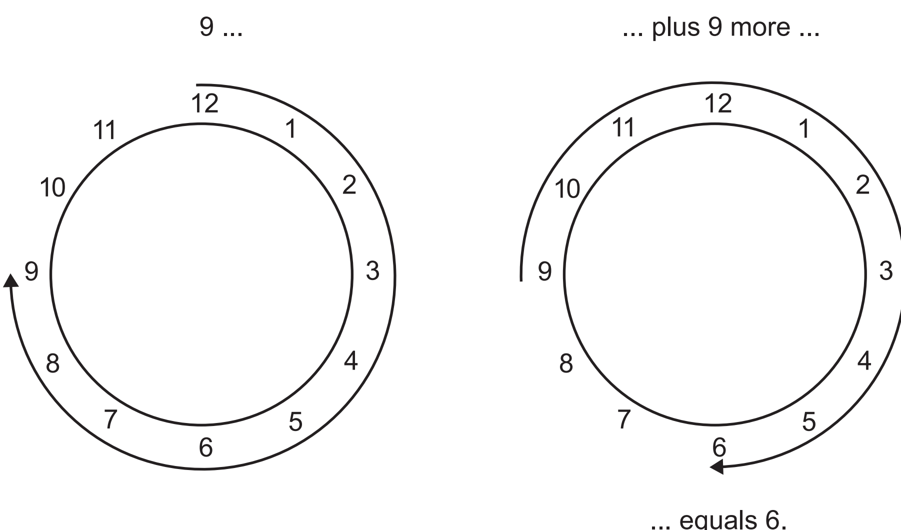
"Casts" on page 7-17

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:

$$\begin{array}{r} 00101 \\ +1 \\ \hline 00110 \end{array} \ (6)$$

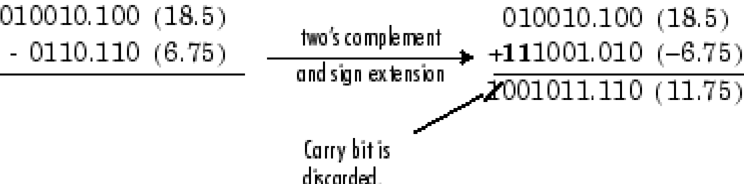
Addition and Subtraction

The addition of fixed-point numbers requires that the binary points of the addends be aligned. The addition is then performed using binary arithmetic so that no number other than 0 or 1 is used.

For example, consider the addition of 010010.1 (18.5) with 0110.110 (6.75):

$$\begin{array}{rr} 010010.1 & (18.5) \\ +0110.110 & (6.75) \\ \hline 011001.010 & (25.25) \end{array}$$

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):

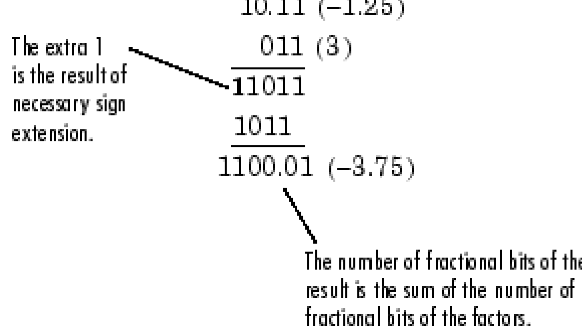


Most fixed-point Signal Processing Blockset blocks that perform addition cast the adder inputs to an accumulator data type before performing the addition. Therefore, no further shifting is necessary during the addition to line up the binary points. See "Casts" on page 7-17 for more information.

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):



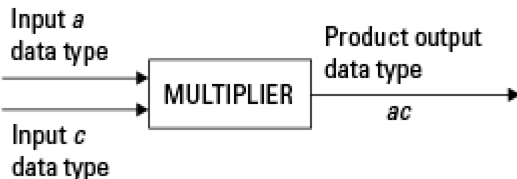
Multiplication Data Types

The following diagrams show the data types used for fixed-point multiplication in Signal Processing Blockset software. The diagrams illustrate the differences between the data types used for real-real, complex-real, and complex-complex multiplication. See individual reference pages in the Block Reference to determine whether a particular block accepts complex fixed-point inputs.

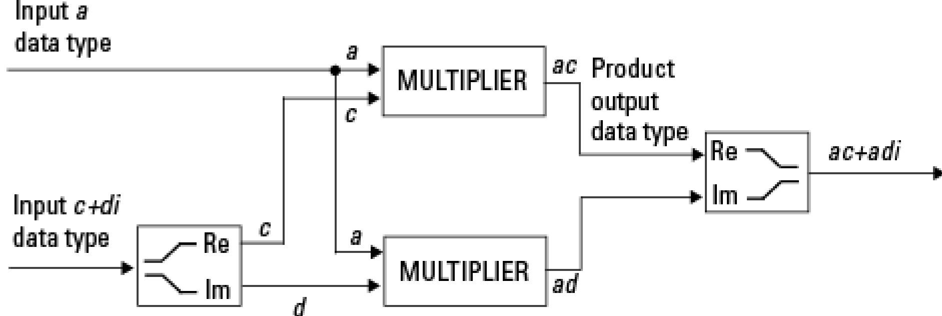
In most cases, you can set the data types used during multiplication in the block mask. See Accumulator Parameters, Intermediate Product ParametersProduct Output Parameters, and Output Parameters. These data types are defined in “Casts” on page 7-17.

Note The following diagrams show the use of fixed-point data types in multiplication in Signal Processing Blockset software. They do not represent actual subsystems used by Signal Processing Blockset software to perform multiplication.

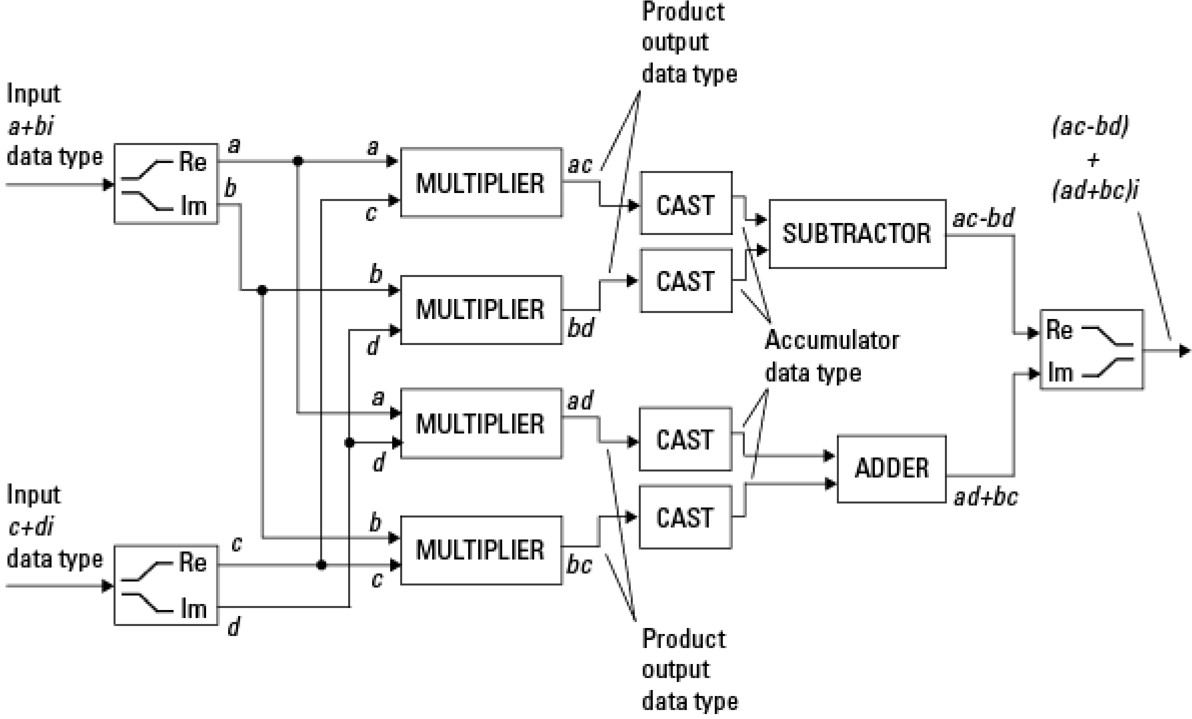
Real-Real Multiplication. The following diagram shows the data types used in the multiplication of two real numbers in Signal Processing Blockset software. The software returns the output of this operation in the product output data type, as the next figure shows.



Real-Complex Multiplication. The following diagram shows the data types used in the multiplication of a real and a complex fixed-point number in Signal Processing Blockset 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 Signal Processing Blockset software. Note that the software returns the output of this operation in the accumulator output data type, as the next figure shows.



Signal Processing Blockset 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 Signal Processing Blockset 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 Signal Processing Blockset 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 Signal Processing Blockset 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

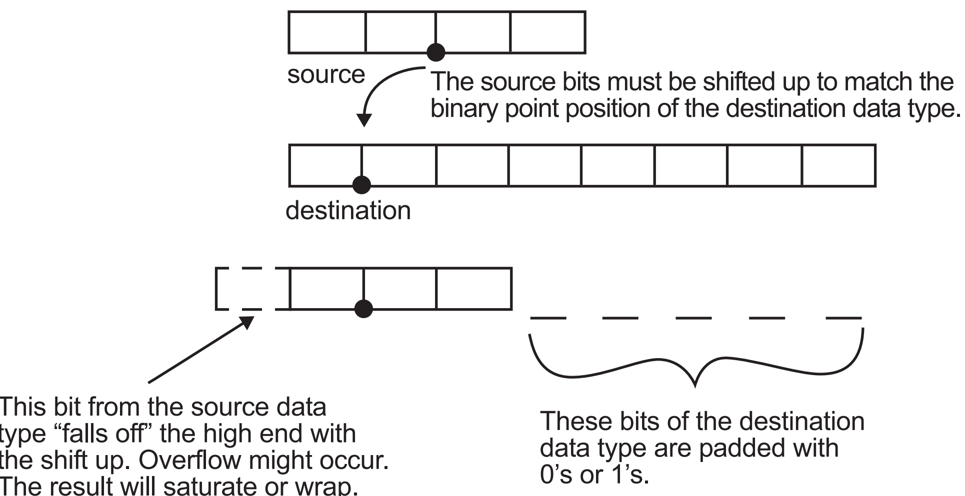
Many fixed-point Signal Processing Blockset blocks allow you to specify the data type and scaling of the block output on the mask. Remember that Signal Processing Blockset software does not allow mixed types on the input and output ports of its blocks. Therefore, if you would like to specify a fixed-point output data type and scaling for a Signal Processing Blockset 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 Signal Processing Blockset 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 Signal Processing Blockset 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.

Casting 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:



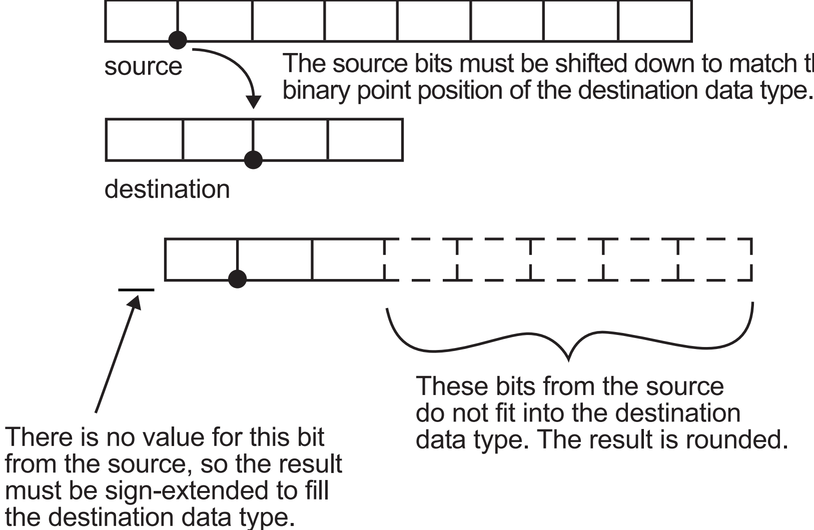
As the diagram shows, the source bits are shifted up so that the binary point matches the destination binary point position. The highest source bit does not fit, so overflow might occur and the result can saturate or wrap. The empty bits at the low end of the destination data type are padded with either 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.

Casting 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.

Specifying Fixed-Point Attributes

In this section...

"Fixed-Point Block Parameters" on page 7-22

"Specifying System-Level Settings" on page 7-25

"Inherit via Internal Rule" on page 7-26

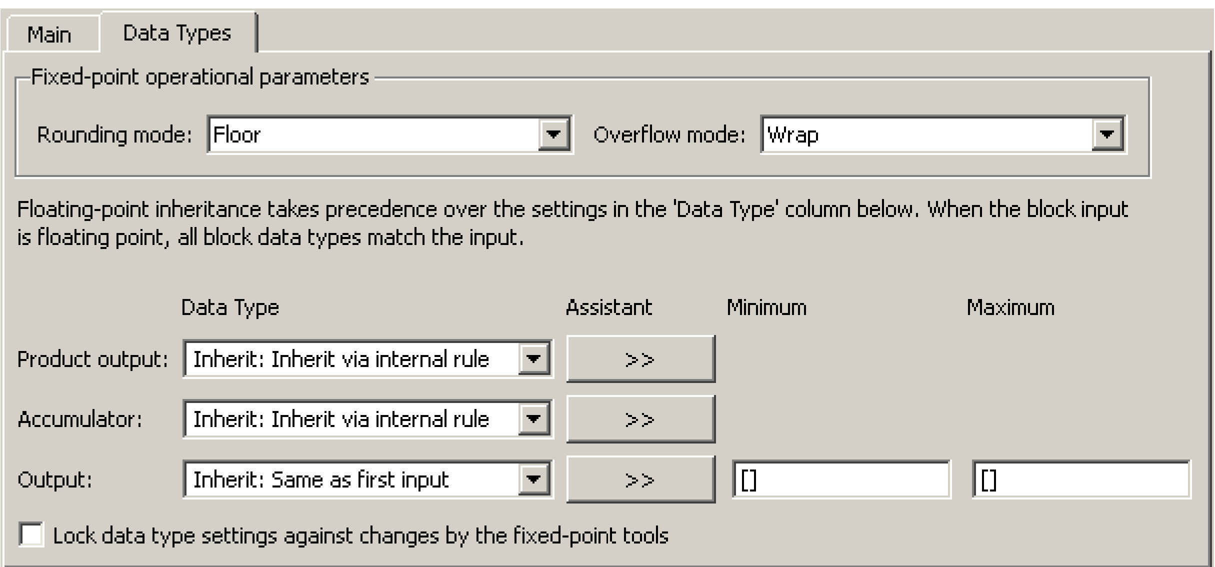
"Example: Selecting and Specifying Data Types for Fixed-Point Blocks" on page 7-37

Fixed-Point Block Parameters

Signal Processing Blockset 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 Signal Processing Blockset blocks. The following figure shows a typical **Data Types** pane.



All Signal Processing Blockset 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 7-9 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 7-8 for more information on the available options.

Fixed-Point Data Type Parameter	Description
Intermediate Product	Specifies the data type and scaling of the intermediate product for fixed-point Signal Processing Blockset blocks. Blocks that feed multiplication results back to the input of the multiplier use the intermediate product data type. See the reference page of a specific block in the Block Reference to learn about the intermediate product data type for that block.
Product Output	Specifies the data type and scaling of the product output for fixed-point Signal Processing Blockset blocks that must compute multiplication results. See the reference page of a specific block in the Block Reference 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 7-14 for more information on complex fixed-point multiplication in Signal Processing Blockset software.
Accumulator	Specifies the data type and scaling of the accumulator (sum) for fixed-point Signal Processing Blockset blocks that must hold summation results for further calculation. Most such blocks cast to the accumulator data type before performing the add operations (summation). See the reference page of a specific block in the Block Reference for details on the accumulator data type of that block.
Output	Specifies the output data type and scaling for Signal Processing Blockset blocks.

Using the Data Type Assistant

The **Data Type Assistant** is an interactive graphical tool available on the **Data Types** pane of some fixed-point Signal Processing Blockset 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 Signal Processing Blockset 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 "Checking Signal Ranges" in the Simulink documentation.

Specifying System-Level Settings

You can monitor and control fixed-point settings for Signal Processing Blockset 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 Signal Processing Blockset 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 Signal Processing Blockset fixed-point data types.

Data type override

Signal Processing Blockset blocks obey the `Use local settings`, `True doubles`, `True singles`, and `Force off` modes of the **Data type override** parameter in the Fixed-Point Tool. The `Scaled doubles` mode is also supported for Signal Processing Blockset 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 Signal Processing Blockset software:

- "Internal Rule for Accumulator Data Types" on page 7-26
- "Internal Rule for Product Data Types" on page 7-27
- "Internal Rule for Output Data Types" on page 7-28
- "The Effect of the Hardware Implementation Pane on the Internal Rule" on page 7-28
- "Internal Rule Examples" on page 7-30

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 7-28 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 7-28 for more information.

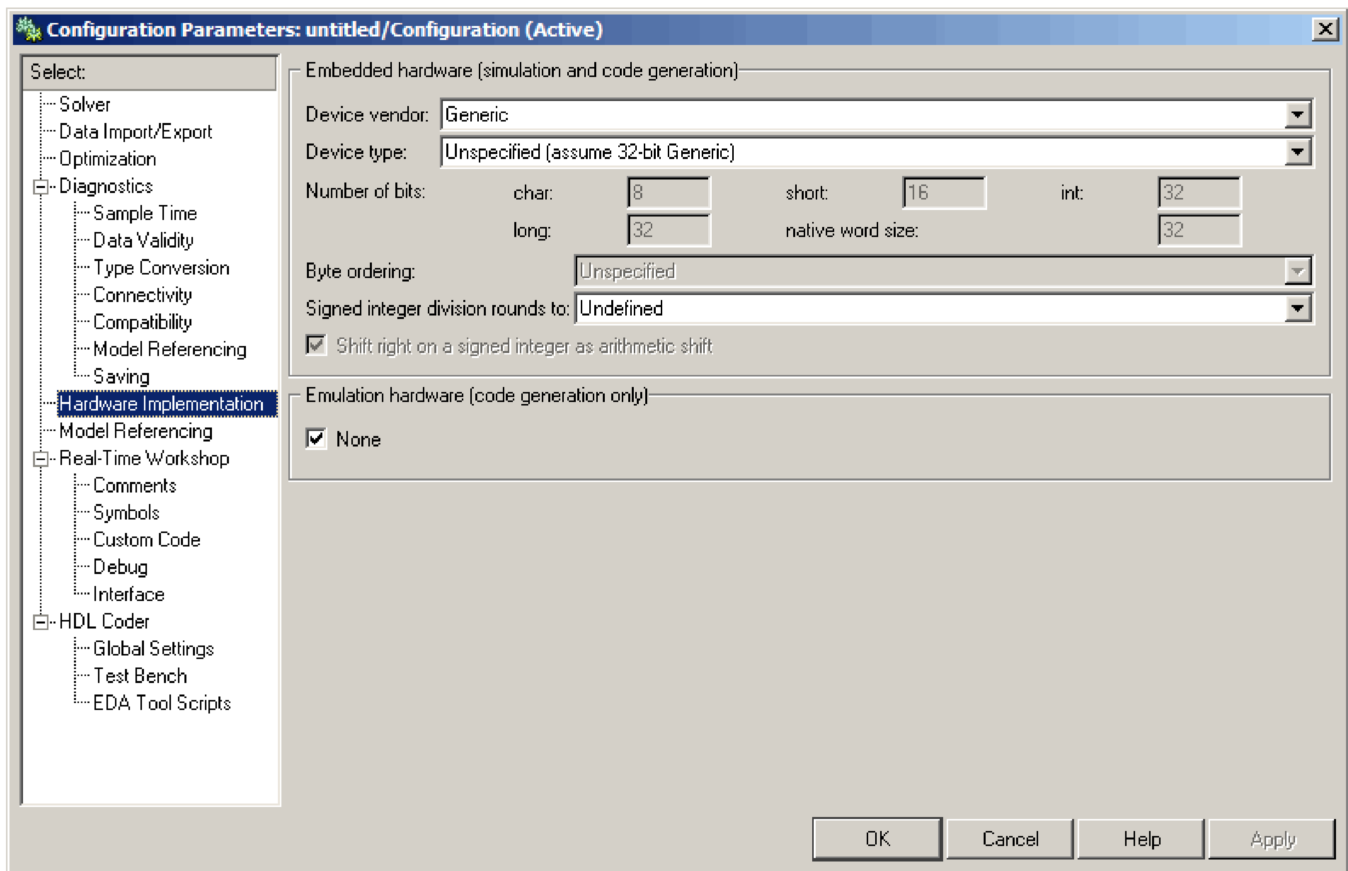
Internal Rule for Output Data Types

A few Signal Processing Blockset 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. For examples, refer to the FFT, IFFT, DCT, and IDCT reference pages.

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 7-28.

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 Signal Processing Blockset software is 128 bits.

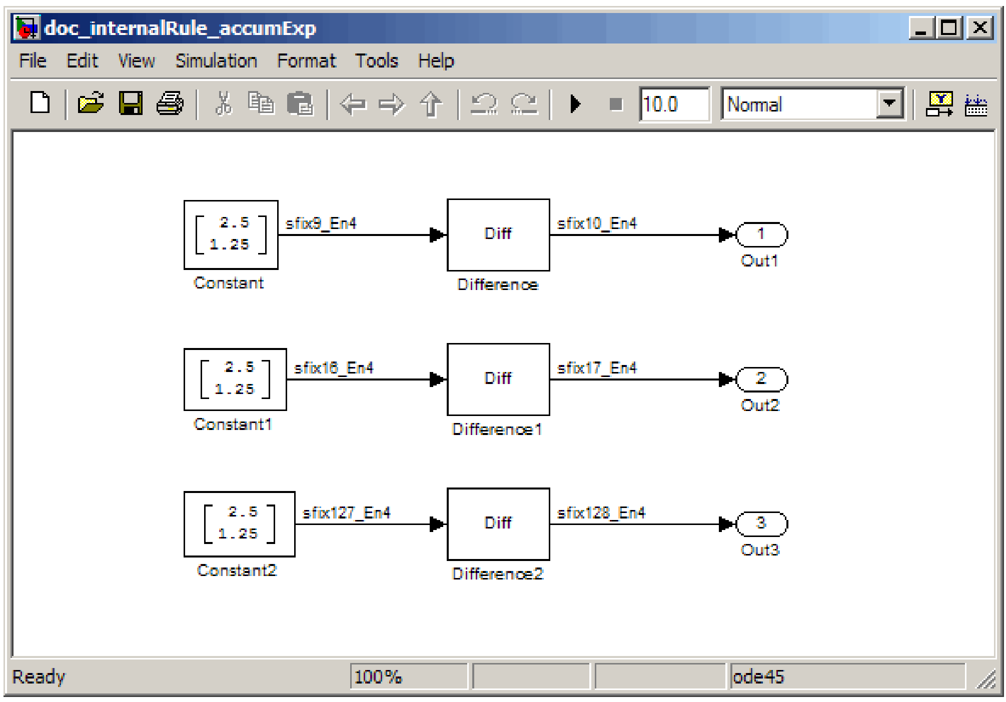
Other targets. For all targets other than ASIC/FPGA, the ideal, full-precision word length calculated by the internal rule is rounded up to the next available word length of the target. The calculated ideal fraction length is used, keeping the least-significant bits.

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 doc_internalRule_accumExp.



In the Difference blocks, the **Accumulator** parameter is set to `Inherit: Inherit via internal rule`, and the **Output** parameter is set to `Inherit:`

`Same as accumulator`. Therefore, you can see the accumulator data type calculated by the internal rule on the output signal in the model.

In the preceding model, the **Device type** parameter in the **Hardware Implementation** pane of the Configuration Parameters dialog box is set to `ASIC/FPGA`. Therefore, the accumulator data type used by the internal rule is the ideal, full-precision result.

Calculate the full-precision word length for each of the Difference blocks in the model:

$$WL_{ideal\ accumulator} = WL_{input\ to\ accumulator} + \text{floor}(\log_2(number\ of\ accumulations)) + 1$$
$$WL_{ideal\ accumulator} = 9 + \text{floor}(\log_2(1)) + 1$$
$$WL_{ideal\ accumulator} = 9 + 0 + 1 = 10$$

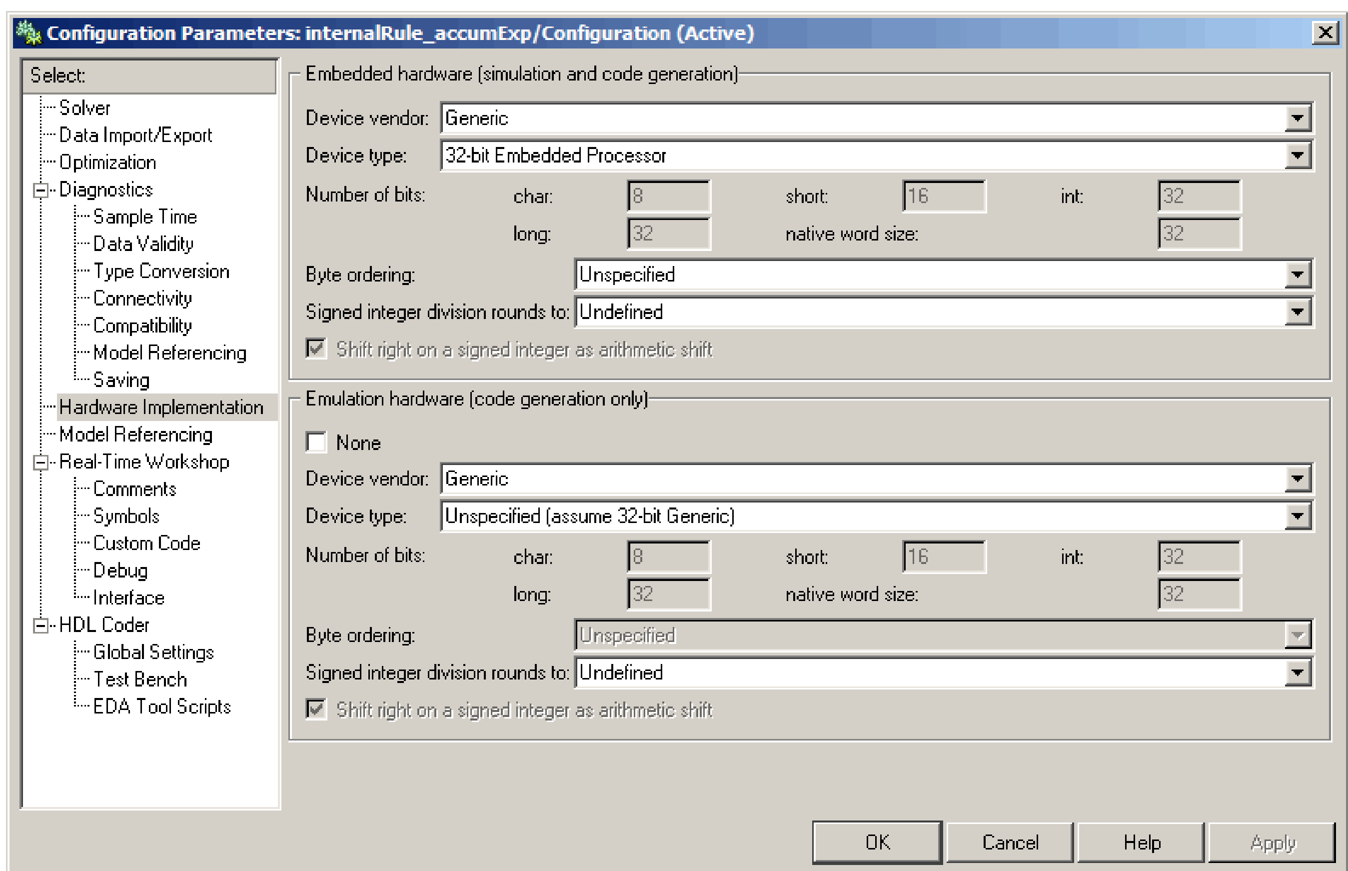
$$WL_{ideal\ accumulator1} = WL_{input\ to\ accumulator1} + \text{floor}(\log_2(number\ of\ accumulations)) + 1$$
$$WL_{ideal\ accumulator1} = 16 + \text{floor}(\log_2(1)) + 1$$
$$WL_{ideal\ accumulator1} = 16 + 0 + 1 = 17$$

$$WL_{ideal\ accumulator2} = WL_{input\ to\ accumulator2} + \text{floor}(\log_2(number\ of\ accumulations)) + 1$$
$$WL_{ideal\ accumulator2} = 127 + \text{floor}(\log_2(1)) + 1$$
$$WL_{ideal\ accumulator2} = 127 + 0 + 1 = 128$$

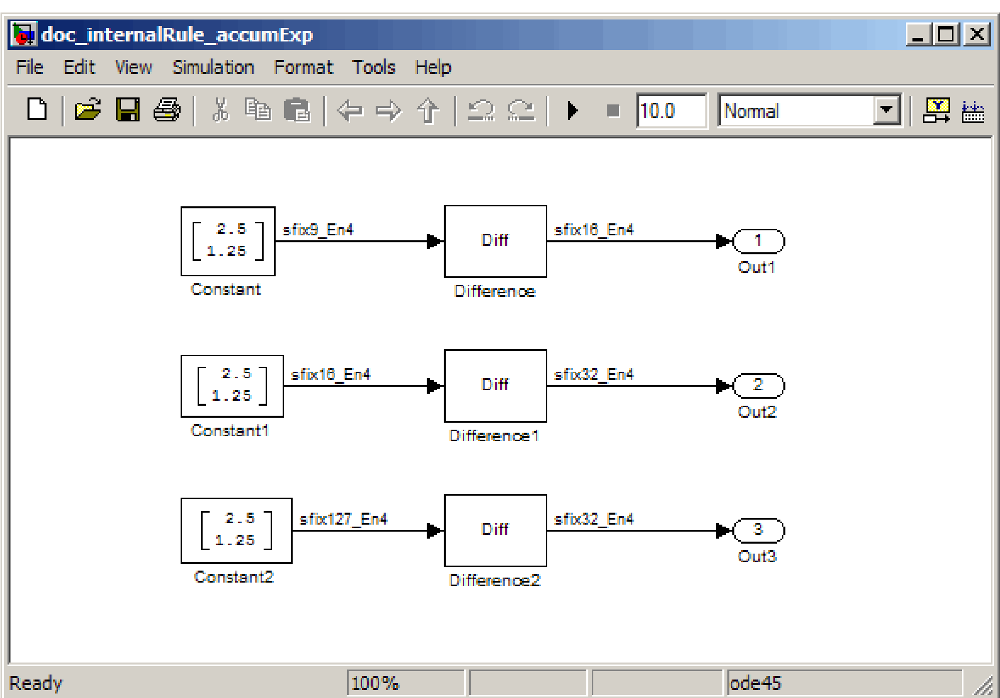
Calculate the full-precision fraction length, which is the same for each Matrix Sum block in this example:

$$FL_{ideal\ accumulator} = FL_{input\ to\ accumulator}$$
$$FL_{ideal\ accumulator} = 4$$

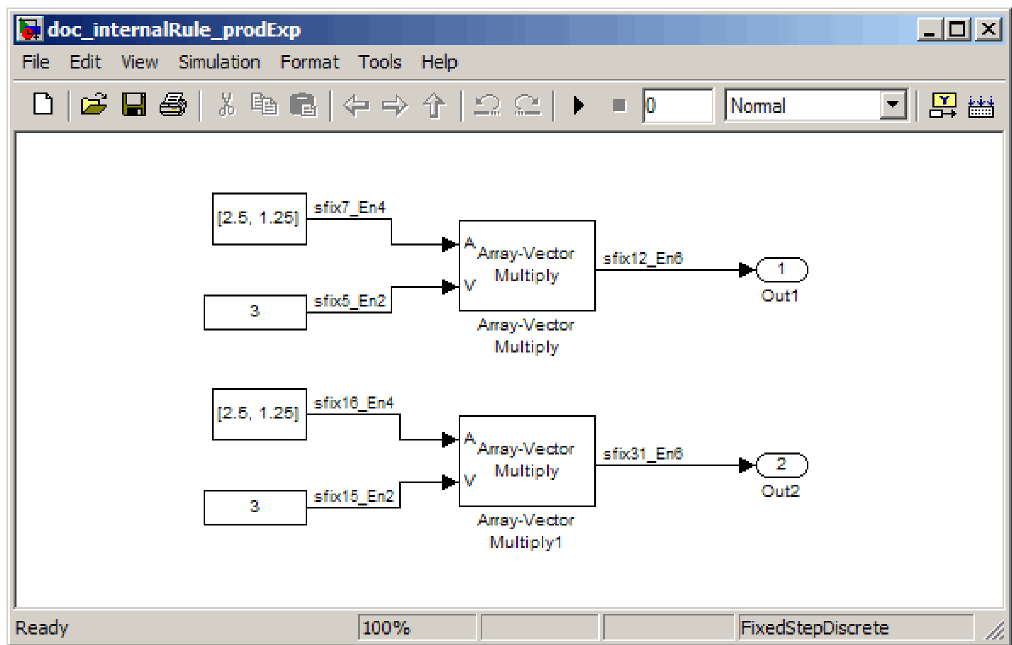
Now change the **Device type** parameter in the **Hardware Implementation** pane of the Configuration Parameters dialog box to `32 bit Embedded Processor`, as shown in the following figure.



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.


doc_internalRule_accumExp
File Edit View Simulation Format Tools Help
10.0
Normal
2.5
1.25
sfix9_En4
Diff
sfix16_En4
1
Constant
Difference
Out1
2.5
1.25
sfix16_En4
Diff
sfix32_En4
2
Constant1
Difference1
Out2
2.5
1.25
sfix127_En4
Diff
sfix32_En4
3
Constant2
Difference2
Out3
Ready
100%
ode45

Product Data Types. Consider the following model doc_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:

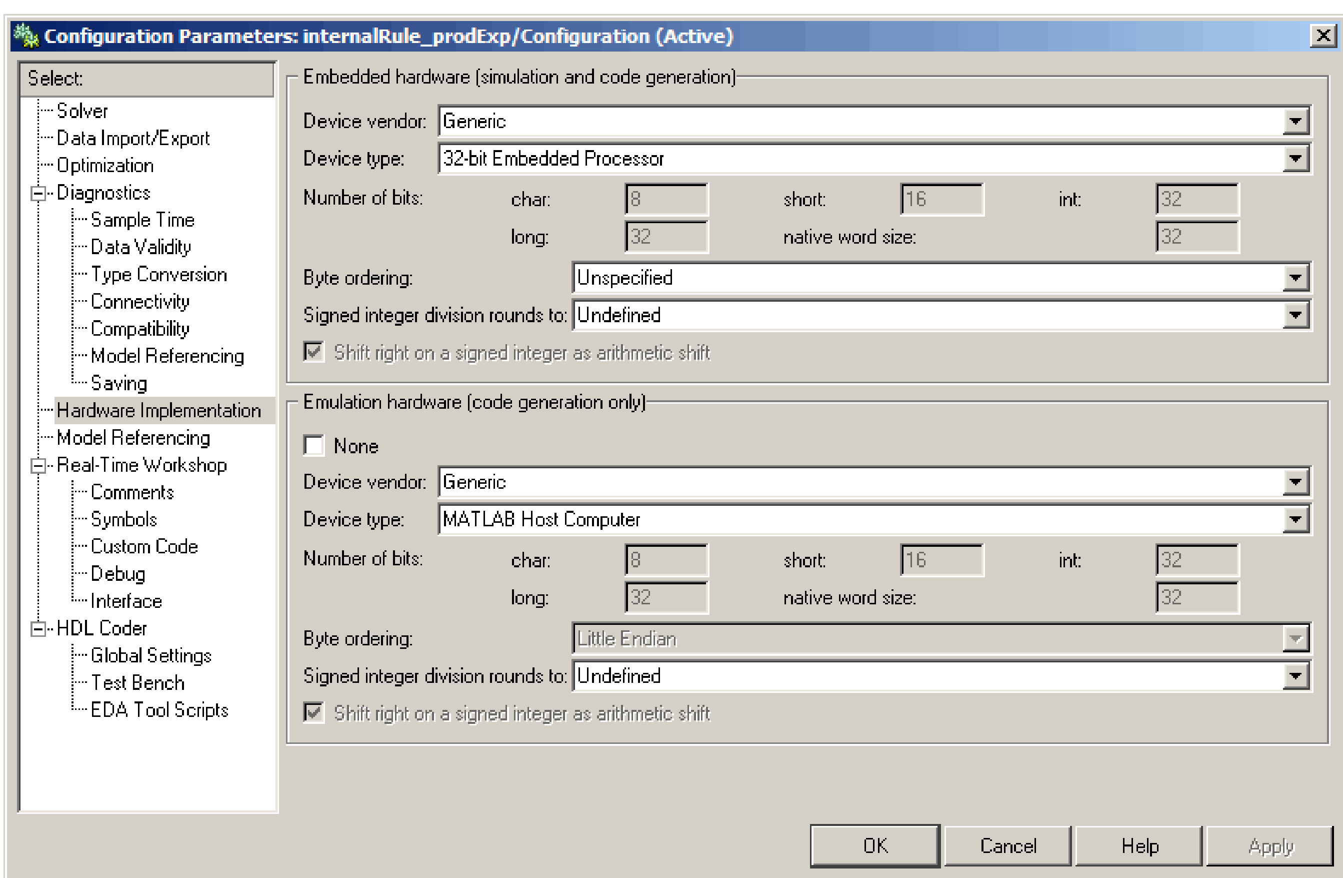
$$WL_{ideal\ product} = WL_{input\ a} + WL_{input\ b}$$
$$WL_{ideal\ product} = 7 + 5 = 12$$

$$WL_{ideal\ product1} = WL_{input\ a} + WL_{input\ b}$$
$$WL_{ideal\ product1} = 16 + 15 = 31$$

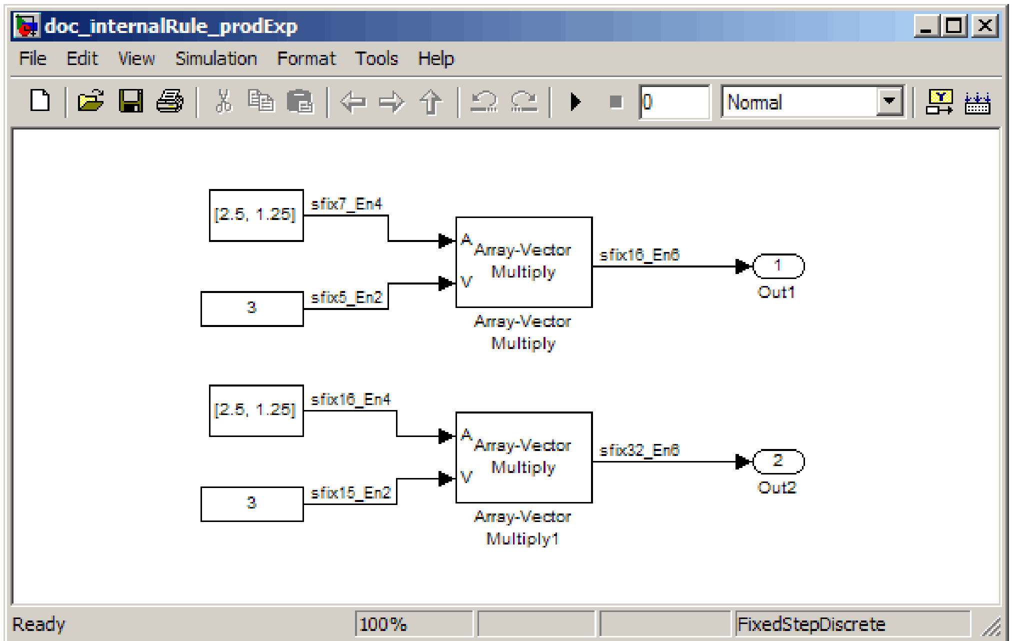
Calculate the full-precision fraction length, which is the same for each Array-Vector Multiply block in this example:

$$FL_{ideal\ product} = FL_{input\ a} + FL_{input\ b}$$
$$FL_{ideal\ product} = 4 + 2 = 6$$

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.



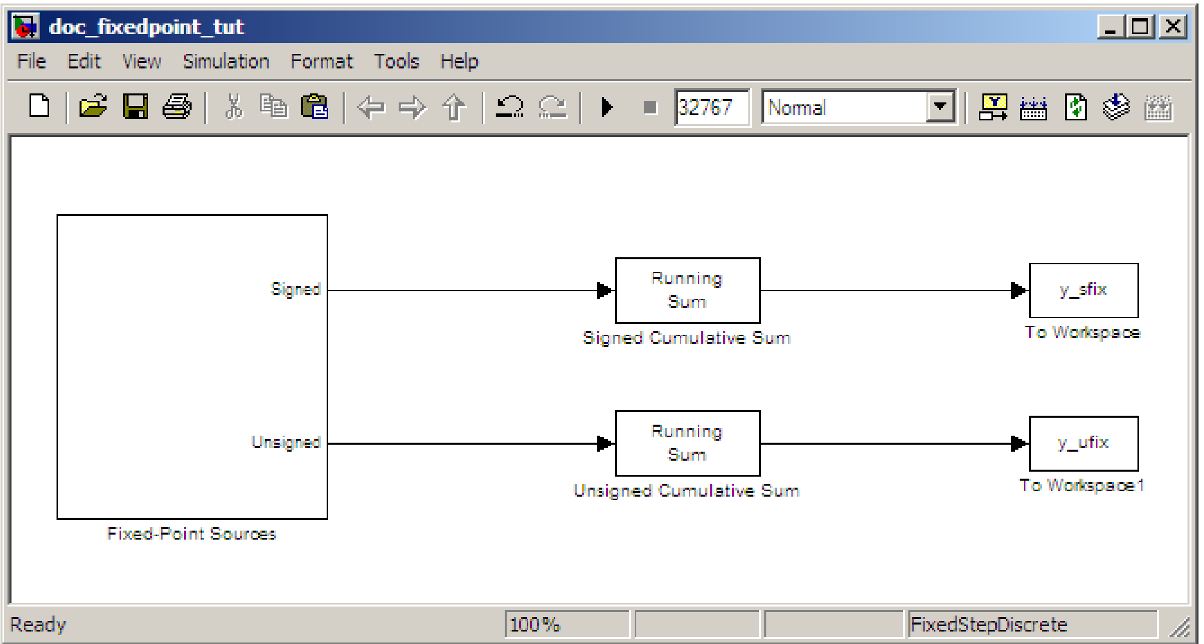
Example: Selecting and Specifying Data Types for Fixed-Point Blocks

The following sections show you how to use the Fixed-Point Tool to select appropriate data types for fixed-point blocks in the doc_dspcumsumfixpt model:

- "Preparing the Model" on page 7-38
- "Using Data Type Override to Find a Floating-Point Benchmark" on page 7-42
- "Using the Fixed-Point Tool to Propose Fraction Lengths" on page 7-42
- "Examining the Results and Accepting the Proposed Scaling" on page 7-43

Preparing the Model

1 Open the model by typing doc_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:

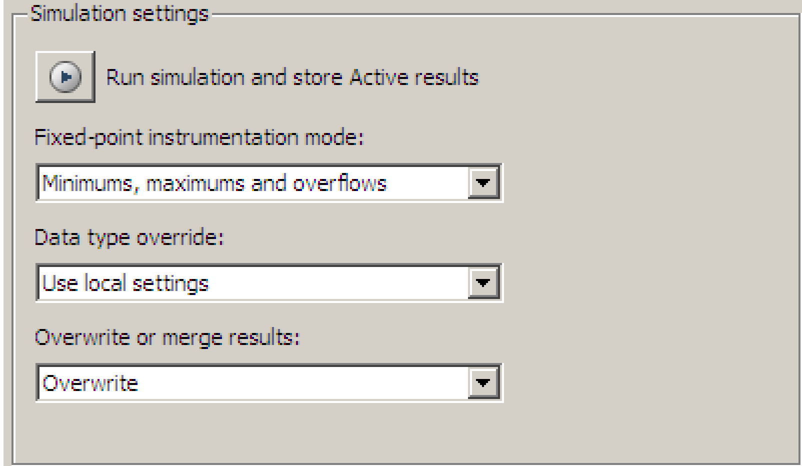
```
Warning: Overflow occurred. This originated from
'doc_fixedpoint_tut/Signed Cumulative Sum'.
```

```
Warning: Overflow occurred. This originated from
'doc_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**.



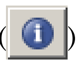
5 The results of the simulation appear in a table in the central **Contents** pane of the Fixed-Point Tool. Review the following columns:

- **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 autoscale information 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 **Contents** 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.

Fixed-point details

Representable maximum: 32767 Assuming signed

Representable minimum: -32768 Assuming signed

Precision: 1

Refresh Details

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 **Run simulation and store active results** 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.

Using 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 `True doubles` 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 `True doubles`.

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 `True doubles` 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.

Using the Fixed-Point Tool to Propose Fraction Lengths

Now that you have your `True doubles` 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 “Checking 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.

Examining the Results and Accepting 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)`).

Fixed-Point Filtering

In this section...

"Fixed-Point Filtering Blocks" on page 7-45
"Filter Implementation Blocks" on page 7-45
"Filter Design and Implementation Blocks" on page 7-46

Fixed-Point Filtering Blocks

The following Signal Processing Blockset 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 Chapter 3, “Filters” and the Filter Realization Wizard reference page in the Block Reference 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.

Getting Started with System Objects

- "What Are System Objects?" on page 8-2
- "Setting Up and Running System Objects" on page 8-3
- "Using System Objects with the Embedded MATLAB Subset" on page 8-9

What Are System Objects?

System objects are MATLAB object-oriented implementations of algorithms. They extend MATLAB by enabling you to model dynamic systems represented by time-varying algorithms. System objects are well integrated into the MATLABlanguage, regardless of whether you are writing simple functions, working interactively in the command window, or creating large applications.

In contrast to MATLAB functions, System objects automatically manage state information, data indexing, and buffering, which is particularly useful for iterative or stream data processing. This enables efficient processing of long data sets. System objects support fixed-point arithmetic and C-code generation from MATLAB and Simulink. With System objects, you can optionally generate code to target the desktop or external hardware. System objects are part of the Embedded MATLAB® subset, and therefore can be used in Simulink® models via the Embedded MATLAB function block.

Setting Up and Running System Objects

In this section...

"Creating an Instance of a System Object" on page 8-3
"Using Methods to Run System Objects " on page 8-6
"Finding Help and Demos for System Objects" on page 8-8

Creating an Instance of a System Object

You must create an instance of a System object before using it. You can create the object at the MATLAB command line or within a program file. The general syntax for creating an instance of a System object with default property values is:

```
<handleName> = <packagename>.<ObjectName>
```

where:

- `handleName` is a MATLAB variable that holds the handle pointing to the created object. System objects are handle objects and follow handle semantics (e.g., when you call a method using the handle, it affects the original object, not a copy of that object). See "The Handle Superclass" for information on handle objects.See "Value or Handle Class — Which to Use" in the MATLAB user documentation for information on object handles.
- `packagename` is the package that contains the particular object. *Packages* are libraries of System objects. For example, these packages implement object versions of associated algorithms.
  - `signalblks` – a package that corresponds to the Signal Processing Blockset
  - `video` – a package that corresponds to the Video and Image Processing Blockset™
- `ObjectName` is the particular object in the package.

This example creates a digital filter object, with default property values, from the `signalblks` package:

```
h = signalblks.DigitalFilter
```

Your command-line code and programs can pass MATLAB variables into and out of System objects.

Understanding System Object Modes

System objects are in one of two modes: *unlocked* or *locked*. After you create an instance of an object and until it starts processing data, that object is in unlocked mode. You can change any of its properties as desired.

When the object begins processing data, it initializes and is locked. When the object is locked, you cannot change the number of inputs or outputs or the value of any nontunable property. You also cannot change the input data type, dimensions, or complexity of any tunable or nontunable property. These restrictions allow the object to maintain states and allocate memory appropriately. The typical way in which an object becomes locked is when the `step` method is called on that object. See "Changing Properties While Running System Objects" on page 8-5 for information on tunable and nontunable properties.

Viewing System Object Properties

System objects have properties that configure the object. You use the default values or set each property to a specific value. The combination of a property and its value is referred to as a *property-value pair*. You can display the list of relevant properties and their current values for an object by using the object handle only, `<handleName>`. Some properties are relevant only when you set another property or properties to particular values. If a property is not relevant, it does not display.

To display a particular property value, use the handle of the created object followed by the property name: `<handleName>.<propertyName>`.

The following code gets the `TransferFunction` property value for the previously created `DigitalFilter` object:

```
h.TransferFunction
```

Setting System Object Property Values

You set the property values of a System object to model the desired algorithm. To set a property when you first create the object, use property-value pair syntax. For properties that allow a specific set of string values, you can use tab completion to select from a list of valid values.

```
<handleName> = <packageName>.<objectName>(property1,value1,...
               property2,value2...)

h1 = signalblks.DigitalFilter('TransferFunction','FIR (all zeros)')
```

To set a property after you create an object, use either of the following syntaxes:

```
<handleName>.<propertyName> = <propertyValue>

h1.TransferFunction = 'FIR (all zeros)'
```

or

```
set(<handleName>,<propertyName>,<propertyValue>)

set(h1,'TransferFunction','FIR (all zeros)'
```

You can enter property-value pairs in any order, except if you are using value-only inputs. Some object properties have no useful default values or are specified every time you create an instance of an object. For these properties, you can specify only the value without specifying the corresponding property name.

If you use value-only inputs, those inputs must be in a specific order. Refer to the object reference page for details. For example, `h2 = signalblks.FIRDecimator(3,fir1(20,0.5))` specifies the decimation factor as `3` and the numerator as `fir1(20,0.5)`.

Changing Properties While Running System Objects

When an object is in locked mode, it is processing data and you can only change the values of properties that are *tunable*. To determine if a particular System object property is tunable, see the corresponding reference page or use help.<packageName>.<objectName>.<propertyName>. For information

on locked and unlocked modes, see "Understanding System Object Modes" on page 8-4.

For most objects, if you change a nontunable property while the object is in locked mode, the object unlocks, loses its state information, and stops processing. For a locked object, if you change the data type, dimensions or complexity of an input or tunable property, the object reinitializes the next time you call the step method. See the object's reference page for more information.

Using Methods to Run System Objects

After you create a System object, you use various object methods to obtain information from the object or have it process data. As an extension of MATLAB classes, most System objects include a standardized set of methods. Some of these methods only apply to particular objects (see "Common Methods " on page 8-7). All methods that are applicable to an object are described in the reference pages for that object.

System object method names begin with a lowercase letter and class and property names begin with an uppercase letter. The syntax for using methods is `<methodName>(<handleName>)`.

Understanding the Advantages of Using Methods

System objects use two commands to process data—a constructor to create the object and a method to run data through the object. This separation of declaration from execution lets you create multiple, persistent, reusable instances of an object, each with different settings. Using this approach avoids repeated input validation and verification, allows for easy use within a programming loop, and improves overall performance. MATLAB functions must validate parameters every time you call the function.

These advantages make System objects particularly well suited for processing streaming data, where segments of a continuous data stream are processed iteratively. This ability to process streaming data provides the advantage of not having to hold large amounts of data in memory. Use of streaming data also allows you to use simplified programs that use loops efficiently.

Common Methods

System objects support the following methods, each of which is described in a method reference page associated with the object. In cases where a method is not applicable to a particular object, calling that method has no effect on the object.

- `step` – Initializes needed resources, processes inputs to the object based on the current object states and properties, returns outputs, and updates the object states. After you call the `step` method, you cannot change any input specifications (i.e., dimensions, data type, complexity). During execution, you can change only tunable properties. The `step` method returns regular MATLAB variables.

  Note For System objects that perform calculations on data (and not just rearrange data), if you pass non-floating point data into that object, it outputs a fixed-point numeric `fi` object, even if you specify the output data type to be to same as the input data type.

  When you create a source object, you specify whether `step` processes the data as samples or as frames. If the object is a source, `step` produces outputs but has no inputs. If the object is a sink, `step` requires inputs, but produces no outputs. For all other objects, `step` has both inputs and outputs, `[out1,...,outN] = step(h,in1,...,inM)`.

- `reset` – Resets the internal states of the object to the initial values for that object.

- `getNumInputs` – Returns the number of inputs expected by the step method. This number varies for an object depending on whether any properties enable additional inputs.

- `getNumOutputs` – Returns the number of outputs from the step method. This number varies for an object depending on whether any properties enable additional outputs.

- `isDone` – Applies only to source objects with end-of-data capability. `isDone` returns logical `true` when the most recent `step` call reaches the end-of-data state. This method returns `false` if either end-of-data is not reached or if the source object does not have end-of-data capability

- `close` – Applies to sink and source objects only. `close` releases any special resources allocated by the object, such as file handles and device drivers.

Finding Help and Demos for System Objects

Refer to the following resources for more information about System objects:

- `help <packagename>` – Lists all System objects in the package, organized by category.
- `help <packagename>.<ObjectName>` — Displays help for the object.
- `doc <packagename>.<ObjectName>` – Displays the reference page for the object, including its properties.
- `help <packageName>.<ObjectName>.<PropertyName>` – Displays help for the property
- `help <packageName>.<ObjectName>.helpFixedPoint` – Displays a list of fixed–point properties for the object.
- `help <packagename>.<ObjectName>.<methodName>` – Displays the method reference page for the object.
- Demos – Provides System object related demos. To view demos, go to online Help contents for the associated product blockset. Under Demos, select MATLAB demos.
- *Object-Oriented Programming* in the MATLAB user documentation – Provides general information about working with objects.

Using System Objects with the Embedded MATLAB Subset

In this section...

"Considerations for Using System Objects with the Embedded MATLAB Subset" on page 8-9

"Using System Objects with Embedded MATLAB Coder" on page 8-11

"Using System Objects with the Embedded MATLAB Function Block" on page 8-12

"Using System Objects with Embedded MATLAB MEX" on page 8-12

Considerations for Using System Objects with the Embedded MATLAB Subset

You can use System objects in code generated using the Embedded MATLAB subset, which is part of the MATLAB language. To generate code, you must also have Simulinkand Real-Time Workshop products.Embedded MATLAB lets you generate efficient code for deployment in embedded systems. It also accelerates fixed-point algorithms. System objects support code generation using the Embedded MATLAB function block in Simulink and using the Embedded MATLAB coder function.

For general information on using Embedded MATLAB, see

- Working with the Embedded MATLAB Subset.
- *Embedded MATLAB Getting Started Guide*.
- *Real-Time Workshop® Embedded Coder™ Getting Started Guide*.

You can customize your generated code by using a configuration object, which is described in "Configuring Your Environment for Code Generation".

The following example, which uses System objects, shows the key factors to consider when you write MATLAB code to be generated using Embedded MATLAB.

```
function lmssystemidentification
% LMSSYSTEMIDENTIFICATION System identification using
```

```
% LMS adaptive filter
%#eml

    % Declare System objects as persistent to generate code
    % using Embedded MATLAB.

    persistent hlms hfilt;

    % Initialize persistent System objects in Embedded MATLAB
    % only once. Do this with 'if isempty(persistent variable).'
    % This condition will be false after the first time.

    if isempty(hlms)

        % Create LMS adaptive filter used for system
        % identification. Pass property value arguments
        % as constructor arguments. Property values must
        % be constants during compile time.

        hlms = signalblks.LMSFilter(11, 'StepSize', 0.01);

        % Create system (an FIR filter) to be identified.

        hfilt = signalblks.DigitalFilter(...
                    'TransferFunction', 'FIR (all zeros)', ...
                    'Numerator', fir1(10, .25));
    end

    x = randn(1000,1);                             % Input signal
    d = step(hfilt, x) + 0.01*randn(1000,1);    % Desired signal
    [~,~,w] = step(hlms, x, d);                    % Filter weights

    % Declare functions called into MATLAB that do not generate
    % code as extrinsic.

    eml.extrinsic('stem');

    stem([get(hfilt, 'Numerator').', w]);
end
```

```
% To compile this function use emlc lmssystemidentification.
% This produces a mex file with the same name in the current
% directory.
```

Review the following considerations when you create code that includes System objects for use with the Embedded MATLAB subset .

- Assign System objects to `persistent` variables.
- Initialize System objects once by embedding the object handles in an `if` statement with a call to `isempty( )`.
- Call the constructor exactly once for any instance of a System object.
- Arguments to System object constructors must be compile-time constants.
- Use the object constructor to set System object properties because Embedded MATLAB does not allow you to use dot notation. Do not `set` any properties during code generation. You can use `get` to display properties.
- Set System object properties using parameter-value pairs only. Do not use value-only inputs.
- Ensure that input to a System object is consistent with the object size, type, and complexity.
- Do not set System objects to become outputs from the Embedded MATLAB function block or from a MEX function generated by Embedded MATLAB.
- Do not pass a System object as an example input argument to a function being compiled with Embedded MATLAB Coder.
- Do not pass a System object from within Embedded MATLAB to functions declared as extrinsic (i.e., functions called in interpreted mode) using `eml.extrinsic`. Do not return System objects from any extrinsic functions.

Using System Objects with Embedded MATLAB Coder

Embedded MATLAB Coder (emlc) is a Real-Time Workshop function that converts MATLAB code into C-code. You can include System objects in a MATLAB program in the same way you include any other program elements. For more information on Embedded MATLAB Coder, see “Converting MATLAB Code to C/C++ Code” and “Generating C Code Using emlc”.

Using System Objects with the Embedded MATLAB Function Block

Using the Embedded MATLAB Function block, you can include a MATLAB language function in a Simulink model. This model can then generate embeddable code using the Embedded MATLAB subset. You can include any System object in the Embedded MATLAB function block. System objects provide higher level algorithms for code generation than do most associated blockset blocks. For more information about the Embedded MATLAB Function block, see Using the Embedded MATLAB Function Block and the Embedded MATLAB Function block reference page in the Simulink documentation.

Using System Objects with Embedded MATLAB MEX

You can use System objects with Embedded MATLAB MEX (`emlmex`), which is particularly useful if you are using System objects that include fixed-point support. `emlmex` converts MATLAB code to C-MEX code which is optimized specifically to accelerate fixed-point algorithms to compiled C-code speed. For more information, see "Working with Embedded MATLAB MEX" in the Embedded MATLAB language subset documentation.

Using Signal Processing System Objects

- "What Are Signal Processing System Objects?" on page 9-2
- "Generating Code for Signal Processing System Objects" on page 9-3
- "Working with Signals and Fixed-Point Data" on page 9-5
- "Example: Using System Objects in Signal Processing Applications: Filtering an Audio Stream" on page 9-15

What Are Signal Processing System Objects?

Signal processing System objects are object-oriented implementations of signal processing algorithms. This set of System objects is organized in a single package, `signalblks`. Many of these objects correspond to block algorithms in the Signal Processing Blockset. A key difference between blocks and System objects is that you include blocks in Simulink models whereas you include System objects in programs or MATLAB command-line code.

Signal processing System objects provide these advantages.

- Support for code generation of algorithms in MATLAB (see "Generating Code for Signal Processing System Objects" on page 9-3).
- Additional support for fixed-point-capable algorithms in MATLAB (see "Working with Fixed-Point Data" on page 9-10).

Generating Code for Signal Processing System Objects

These signal processing System objects support code generation in MATLAB via Embedded MATLAB Coder (emlc), which requires Simulink and Real-Time Workshop. See "Using System Objects with the Embedded MATLAB Subset" on page 8-9 for information on generating code.

Signal Processing Code Generation Support

```
signalblks.ArrayVectorAdder
signalblks.ArrayVectorDivider
signalblks.ArrayVectorMultiplier
signalblks.ArrayVectorSubtractor
signalblks.BiquadFilter
signalblks.BurgAREstimator
signalblks.CepstralToLPC
signalblks.CholeskyFactorization
signalblks.Convolver
signalblks.Counter
signalblks.Crosscorrelator
signalblks.CumulativeProduct
signalblks.CumulativeSum
signalblks.DCT
signalblks.Delay
signalblks.DelayLine
signalblks.DigitalFilter
signalblks.FFT
signalblks.FIRInterpolator
signalblks.FIRRateConverter
signalblks.Histogram
signalblks.IDCT
signalblks.IFFT
signalblks.Interpolator
signalblks.LMSFilter
signalblks.LPCToAutocorrelation
signalblks.LPCToCepstral
signalblks.LPCToLSF
signalblks.LPCToRC
```

```
signalblks.LSFToLPC
signalblks.LUFactorization
signalblks.LevinsonSolver
signalblks.LowerTriangularSolver
signalblks.Maximum
signalblks.Mean
signalblks.Median
signalblks.Minimum
signalblks.NCO
signalblks.Normalizer
signalblks.PeakFinder
signalblks.PhaseUnwrapper
signalblks.QRFactorization
signalblks.QRSolver
signalblks.RCToLPC
signalblks.RMS
signalblks.ScalarQuantizerDecoder
signalblks.ScalarQuantizerEncoder
signalblks.SineWave
signalblks.StandardDeviation
signalblks.UpperTriangularSolver
signalblks.VariableFractionalDelay
signalblks.VariableIntegerDelay
signalblks.Variance
signalblks.VectorQuantizerDecoder
signalblks.VectorQuantizerEncoder
signalblks.Window
signalblks.ZeroCrossingDetector
```

Working with Signals and Fixed-Point Data

In this section...

"What Are Sample- and Frame-Based Processing?" on page 9-5

"Working with Fixed-Point Data" on page 9-10

What Are Sample- and Frame-Based Processing?

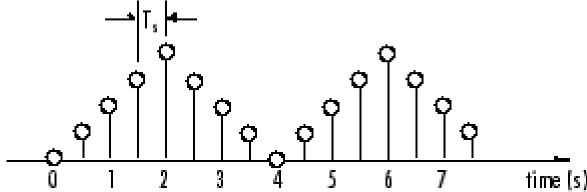
System objects interpret signals as either sample-based or frame-based inputs. Signals can also be either single channel or multichannel. In sample-based processing, the input signal is processed one sample at a time, rather than in batches of samples (see"Sample-Based Single Channel Processing" on page 9-5 and "Sample-Based Multichannel Processing" on page 9-6). In frame-based processing, portions of the input signal are batched into frames and each frame is processed separately (see "Frame-Based Single Channel Processing" on page 9-7 and "Frame-Based Multichannel Processing" on page 9-7).

See "Setting the Sample- or Frame-Based Processing Property" on page 9-9for information on setting the property that controls sample- or frame-based processing.

Note System objects process all multidimensional (n-D) array input, where n > 2, as sample based.

Sample-Based Single Channel Processing

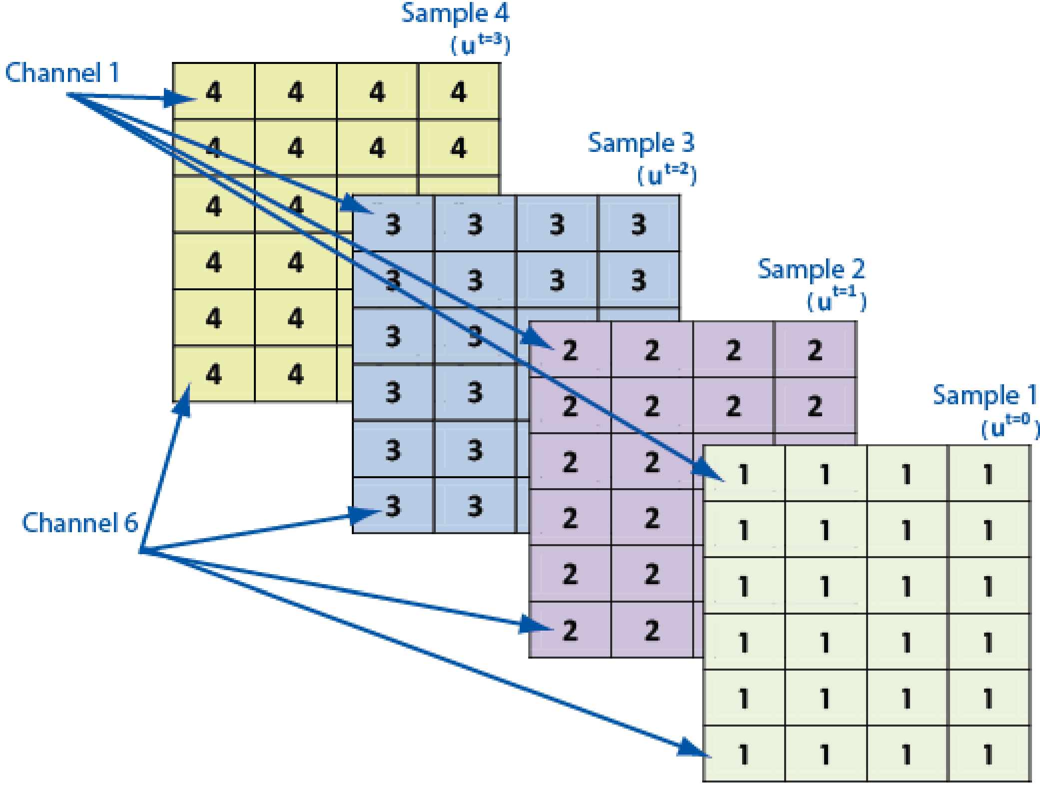
The following figure shows a discrete-time signal. This signal is single channel because it is one independent sequence of numbers. System objects processes this signal as sample-based if each sample is processed separately.



Sample-Based Multichannel Processing

An *M*-by-*N* matrix that a System object processes as sample-based inputs represents *M***N* independent channels. Each matrix element represents one sample from a distinct channel, i.e., each channel contains a single value.

For example, consider the 24-channel (6-by-4) multichannel signal in the next figure, where $u^{t=0}$ is the first matrix in the series, $u^{t=1}$ is the second, etc.



The signal in channel 1 is the following sequence:

$$u_{11}^{t=0}, u_{11}^{t=1}, u_{11}^{t=2}, \ldots$$

Similarly, channel 9 (counting down the columns) contains the following sequence:

$$u_{32}^{t=0}, u_{32}^{t=1}, u_{32}^{t=2}, \ldots$$

When a System object processes this signal as sample-based input, it processes each element in a sequence separately. This type of processing for a large multichannel signal is very expensive and not very practical.

Frame-Based Single Channel Processing

Data samples are typically transmitted in batches, or frames, and not as single elements. For efficiency and to reduce computational costs, most signals are processed as frames, which processes multiple samples at once. A frame of data is a collection of sequential samples from a single channel or multiple channels. One frame of a single-channel signal is represented by an M -by-1 column vector.

Frame-Based Multichannel Processing

One frame of a multichannel signal is represented by an M-by-N matrix. Each matrix column is a different channel, and each row is a sample, containing multiple data points. The number of rows in the matrix represents the number of samples within each frame. Consider a sequence of frame matrices, where $u^{t=0}$ is the first matrix (i.e., first frame) in a series, $u^{t=1}$ is the second, $u^{t=2}$ is the third, etc.

This sequence represents the data in channel 1:

$$u_{11}^{t=0}, u_{21}^{t=0}, u_{31}^{t=0}, \ldots, u_{M1}^{t=0}, u_{11}^{t=1}, u_{21}^{t=1}, u_{31}^{t=1}, \ldots, u_{M1}^{t=1}, u_{11}^{t=2}, u_{21}^{t=2}, \ldots$$

Similarly, this next sequence represents the data in channel 3:

$$u_{13}^{t=0}, u_{23}^{t=0}, u_{33}^{t=0}, \ldots, u_{M3}^{t=0}, u_{13}^{t=1}, u_{23}^{t=1}, u_{33}^{t=1} \ldots, u_{M3}^{t=1}, u_{13}^{t=2}, u_{23}^{t=2}, \ldots$$

When a System object processes a signal as frame-based input, that object processes each frame in a sequence separately.

Setting the Sample- or Frame-Based Processing Property

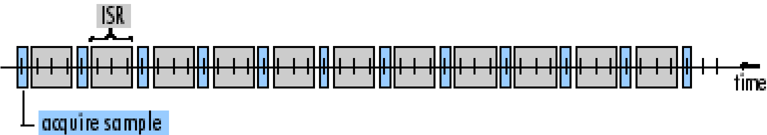
All System objects support sample-based processing and some System objects support both sample- and frame-based processing. To specify how your object should process input data, you set the `FrameBasedProcessing` property. The property has a default value of `true`, which enables frame-based processing.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

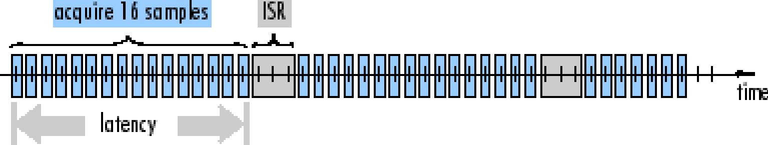
Benefits of Frame-Based Processing

Frame-based processing accelerates both real-time systems and simulations. Real-time systems typically process data in frames. Data acquisition hardware often operates by accumulating a large number of signal samples at a high rate and propagating these samples to the real-time system as a block of data. This operation maximizes the efficiency of the system by distributing the process overhead across many samples. Such distribution allows the relatively fast data acquisition to be interrupted less by the relatively slow data processing after each frame is acquired, rather than after each individual sample.

The next figure illustrates how frame-based data acquisition increases throughput. The thin blocks each represent the time elapsed during acquisition of a sample. The wider 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. The frame-based throughput rate is many times higher than the sample-based alternative.





Working with Fixed-Point Data

- "Getting Information About Fixed-Point System Objects" on page 9-10
- "Displaying Fixed-Point Properties" on page 9-12
- "Setting System Object Fixed-Point Properties" on page 9-13

Getting Information About Fixed-Point System Objects

General information about working with fixed-point data processing is in the Chapter 7, "Working with Fixed-Point Data" topic. This topic refers to models and blocks, but the information and concepts apply to System objects, too. Applicable subtopics are:

- "Fixed-Point Signal Processing Development" on page 7-2 – fixed-point hardware, design and applications
- "Concepts and Terminology" on page 7-5 – data types, scaling, and precision and range
- "Arithmetic Operations" on page 7-11 – modulo, two's complement, addition, subtraction, multiplication, and casts
- "Specifying Fixed-Point Attributes" on page 7-22 – fixed-point parameters (which correspond to System object properties), system-level settings and inheriting via the internal rule

- "Fixed-Point Filtering" on page 7-45 – fixed-point issues for filtering

System objects that support fixed-point data processing have fixed-point properties, which you can display for a particular object by typing `signalblks.<ObjectName>.helpFixedPoint` at the command line. See "Displaying Fixed-Point Properties" on page 9-12 to set the display of System object fixed-point properties.

The following signal processing System objects support fixed-point data processing.

Signal Processing System Objects That Support Fixed-Point Data Processing

```
signalblks.ArrayVectorAdder
signalblks.ArrayVectorDivider
signalblks.ArrayVectorMultiplier
signalblks.ArrayVectorSubtractor
signalblks.Autocorrelator
signalblks.BiquadFilter
signalblks.Convolver
signalblks.Counter
signalblks.Crosscorrelator
signalblks.CumulativeProduct
signalblks.CumulativeSum
signalblks.DCT
signalblks.DigitalFilter
signalblks.FFT
signalblks.FIRDecimator
signalblks.FIRInterpolator
signalblks.FIRRateConverter
signalblks.Histogram
signalblks.IDCT
signalblks.IFFT
signalblks.LDLFactorization
signalblks.LMSFilter
signalblks.LUFactorization
signalblks.LevinsonSolver
```

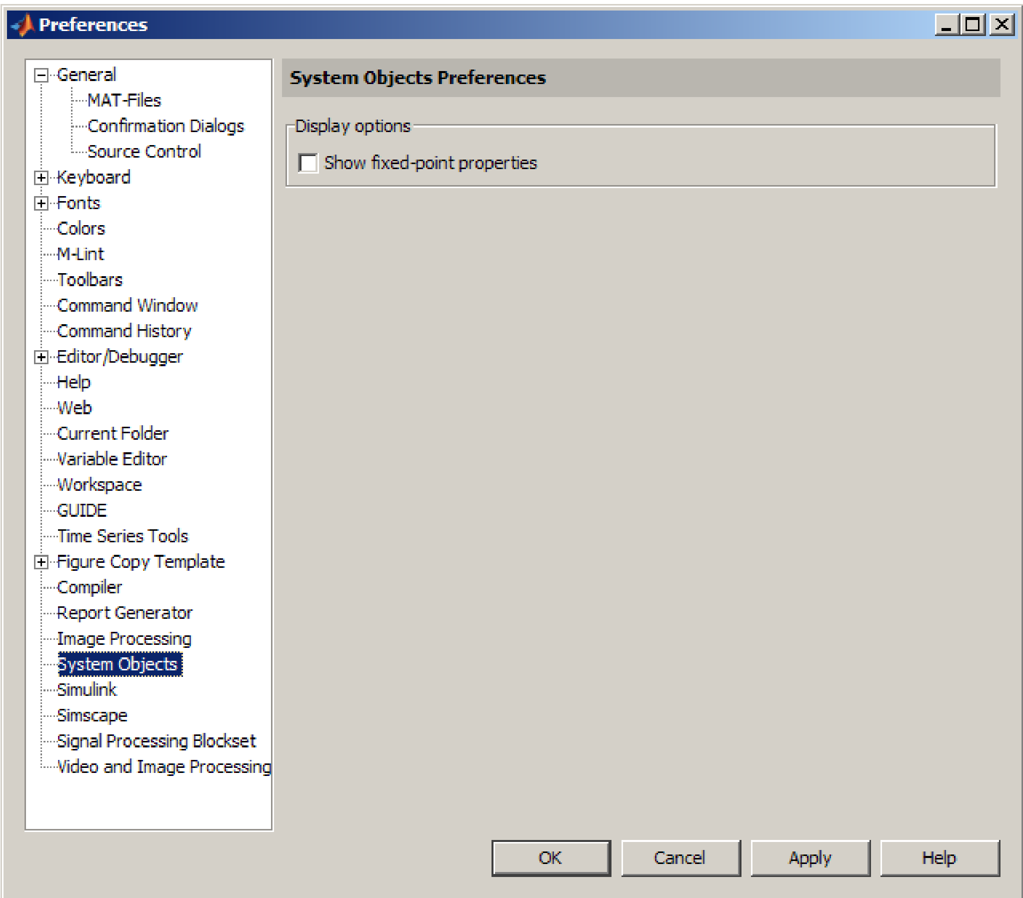
```
signalblks.LowerTriangularSolver
signalblks.Maximum
signalblks.Mean
signalblks.Median
signalblks.Minimum
signalblks.NCO
signalblks.Normalizer
signalblks.PeakFinder
signalblks.ScalarQuantizerDecoder
signalblks.ScalarQuantizerEncoder
signalblks.SineWave
signalblks.SubbandAnalysisFilter
signalblks.SubbandSynthesisFilter
signalblks.UpperTriangularSolver
signalblks.VariableFractionalDelay
signalblks.VariableFractionalDelay
signalblks.Variance
signalblks.VectorQuantizerDecoder
signalblks.VectorQuantizerEncoder
```

Displaying Fixed-Point Properties

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.

Setting 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 "Setting System Object Property

Values" on page 8-5. You also use the Fixed-Point Toolbox`numerictype` object to specify the desired data type as fixed-point, the signedness, and the word- and fraction-lengths. System objects support these values of `DataTypeMode`: `boolean`, `double`, `single`, and `Fixed-point:  binary point scaling`.

In the same way as for blocks, the data type properties of many System objects can set the appropriate word lengths and scalings automatically by inheriting via the internal rule (see "Inherit via Internal Rule" on page 7-26 for how this rule applies to different types of blocks and objects).

In most cases, if you have not set the property that activates a dependent property and you attempt to change that dependent property, a warning message displays. As a convenience, if you set a dependent, fixed-point, `Custom<xxx>DataType` property before setting the `<xxx>DataType` property, the System object automatically sets `<xxx>DataType` for you to activate the dependent property. <xxx> differs for each object. For example, for the `signalblks.FFT` object, setting `CustomOutputDataType` to `numerictype(1,32,30)` automatically sets `OutputDataType` to `'Custom'`.

Example: Using System Objects in Signal Processing Applications: Filtering an Audio Stream

This algorithm reads and filters a long audio stream, and then plays it with an audio player. The `MultimediaFileReader` System object specifies the desired file name directly as the first input argument to the constructor. For efficiency and to provide optimum results, the objects are instantiated outside the while loop.

```
% Create handle reference to object that reads a multimedia file.
hAudioIn  = signalblks.MultimediaFileReader('speech_dft.avi');

% Create handle reference to object for a lowpass FIR filter.
hFiltLP   = signalblks.DigitalFilter(...
  'TransferFunction', 'FIR (all zeros)', ...
  'Numerator',fir1(16,0.375));

% Create handle reference to a component that sends audio
% to the speakers at a specified sampling rate.
hAudioOut = signalblks.AudioPlayer('SampleRate',22050);

% Call the processing loop to read the audio stream, filter it
% and play it. The loop stops when you reach the end of the
% input file, which is detected by the MultimediaFileReader
% component.
while ~isDone(hAudioIn)
 data = step(hAudioIn);
 temp = step(hFiltLP, data);
 step(hAudioOut, temp);
end
% Wait for the audio queue to finish streaming
pause(1);

% Call the close method on components to close files and devices.
close(hAudioIn);
close(hAudioOut);close(hAudioIn);
close(hAudioOut);
```

Separating the instantiation of the objects from the data processing section eliminates the repeated parsing and validation of the input data and property values. Such separation also avoids resetting of algorithm parameters repeatedly in the loop.

Index

A

adaptive filters 3-53
- creating 3-55
- customizing 3-60

add
- samples 2-25

algebraic loop errors 2-58
algorithmic delay 2-51
- adjustable 2-54
- and initial conditions 2-54
- basic 2-54
- excess 2-57
- relation to latency 2-57
- zero 2-51

analog filter designs 3-51
- *See also* filter designs

angular frequency 1-2
- *See also* periods

arithmetic operations
- fixed-point 7-11

arrays
- importing 1-61

attenuation
- stopband 3-51

auto-promoting rates 1-9
avoiding unintended rate conversion 2-19

B

band configurations 3-51
bandpass filter designs
- analog, available parameters 3-51

bandstop filter designs
- analog, available parameters 3-51

basic
- statistical operations 6-3

basic algorithmic delay 2-54
benefits
- frame-based processing 2-50

block parameters
- fixed-point 7-22

block rate types 2-58
blocks
- multirate 2-58
- single-rate 2-58

Buffer overlap parameter
- negative values for 2-37

buffering 2-25
- altering the sample period of the signal 2-30
- altering the signal 2-26
- causing unintentional rate conversions 2-24
- frame-based signals into other frame-based signals 2-41
- internal 2-36
- preserving the sample period of the signal 2-27
- sample-based signals into frame-based signals 2-33
- sample-based signals into frame-based signals with overlap 2-37

`butter` function 3-52
Butterworth filter designs
- analog 3-51
- band configurations for 3-51

C

casts
- fixed-point 7-17

changing
- frame sizes 2-15
- the frame size of a signal 2-27

channels
- of a sample-based signal 1-13

`cheby1` function 3-52
`cheby2` function 3-52
Chebyshev type I filter designs
- analog 3-51
- band configurations for 3-51

Chebyshev type II filter designs

analog 3-51
band configurations for 3-51
choosing
filter design blocks 3-18
close method 8-8
code generation
fixed-point 7-3
generic real-time (GRT) 2-50
signal processing objects 9-3
combining
frame-based signals 1-39
multichannel sample-based signals 1-35
single-channel sample-based signals 1-32
complex multiplication
fixed-point 7-14
computational delay 2-49
reducing 2-49
concatenating
frame-based signals 1-39
multichannel sample-based signals 1-35
single-channel sample-based signals 1-32
concepts
frame rate 2-2
sample rate 2-2
configuring
vector quantization model 5-11
continuous-time
discretizing signals 1-11
signals 1-11
source blocks 1-11
conventions
time and frequency 1-2
converting 2-12
frame rates 2-12
frame-based signals into other frame-based signals 2-41
sample-based signals into frame-based signals 2-33
sample-based signals into frame-based signals with overlap 2-37
See also rate conversion
creating
1-D vector signal 1-20
adaptive filters 3-55
fixed-point filters 3-32
frame-based signals 1-25
multichannel frame-based signals 1-38
multichannel sample-based signals 1-32
sample-based signals 1-19
scalar quantizers 5-5
vector quantizers 5-10
customizing
adaptive filters 3-60

D

deconstructing
multichannel frame-based signals 1-49
multichannel frame-based signals into individual signals 1-49
multichannel sample-based signals 1-42
multichannel sample-based signals into individual signals 1-42
multichannel sample-based signals into other multichannel signals 1-45
delay
algorithmic 2-51
computational 2-49
rebuffering 2-44
relation to latency 2-57
delete
samples 2-25
demos
multirate filtering 3-74
designing
adaptive filters 3-55
fixed-point filters 3-32
scalar quantizers 5-5
vector quantizers 5-10
Digital Filter block

filtering noise with 3-5
Digital Filter Design block
filtering noise with 3-25
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 1-100
line widths 2-12
time-domain data 1-79
downsampling 2-12
See also rate conversion

E

`ellip` function 3-52
elliptic filter designs
analog 3-51
band configurations for 3-51
errors
algebraic loop 2-58
due to continuous-time input to a discrete-time block 1-11
sample-rate mismatch 1-6
estimation
power spectrum 6-6
examples
latency 2-59
multirate filtering 3-74
exporting
frame-based signals 1-73
sample-based signals 1-65

F

factoring matrices 6-9
FFT block
using 4-2
FFT length parameter 2-21
filter band configurations 3-51
filter design blocks
choosing 3-18
filter designs
available parameters 3-51
`butter` function 3-52
Butterworth 3-51
`cheby1` function 3-52
`cheby2` function 3-52
Chebyshev type I 3-51
Chebyshev type II 3-51
continuous-time 3-51
`ellip` function 3-52
elliptic 3-51
passband ripple 3-51
stopband attenuation 3-51
filtering an audio stream
System object example 9-15
filters
adaptive 3-53
creating a highpass filter 3-23
creating a lowpass filter 3-21
Filter Realization Wizard 3-31
filtering noise with Digital Filter blocks 3-5
filtering noise with Digital Filter Design blocks 3-25
fixed-point 7-45
implementing a highpass filter 3-4
implementing a lowpass filter 3-3
multirate 3-66
fixed point
System object preferences 9-12
fixed point properties
System objects 9-13
fixed-point attributes, specification

at the block level 7-22
at the system level 7-25
fixed-point block parameters
setting 7-22
fixed-point code generation 7-3
fixed-point data types 7-5
addition 7-13
arithmetic operations 7-11
attributes 7-22
casts 7-17
complex multiplication 7-14
concepts 7-5
filters 7-45
logging 7-25
modular arithmetic 7-11
multiplication 7-14
overflow handling 7-7
precision 7-7
range 7-7
rounding 7-8
saturation 7-7
scaling 7-6
subtraction 7-13
terminology 7-5
two's complement 7-12
wrapping 7-7
fixed-point development
benefits 7-2
fixed-point DSP applications 7-4
fixed-point filters
designing and implementing 3-32
Fixed-Point Tool 7-25
fixed-step solvers 1-6
frame periods 2-11
altered by unbuffering 2-45
constant 2-12
converting 2-11
multiple 2-12
related to sample period and frame size 2-2
Simulink Probe block 2-5
See also rate conversion
frame rates 1-9
auto-promoting 1-9
color coding 2-9
concepts 2-2
inspecting 2-9
See also frame periods
frame rebuffering
blocks for 2-24
frame sizes 2-11
changing 2-27
constant 2-12
converting 2-11
converting by rebuffering 2-11
direct rate conversion 2-11
maintaining a constant frame rate 2-12
maintaining a constant sample rate 2-25
related to sample period and frame
period 2-2
See also rate conversion
frame-based multichannel processing 9-7
frame-based multichannel signals 1-15
See also signals
frame-based processing 9-5
benefits 2-50
latency 1-17
frame-based signals
benefits of 1-16
combining 1-39
concatenating 1-39
converting to other frame-based signals 2-41
creating 1-25
deconstructing multichannel signals 1-49
exporting 1-73
importing 1-70
importing and exporting 1-70
multichannel 1-15
reordering channels in a multichannel
signal 1-54
separating multichannel signals 1-49

single channel 1-15
unbuffering to sample-based signals 2-45
frame-based single channel processing 9-7
frame-matrices
format of 1-15
frame-rate adjustment
rate conversion 2-12
frame-size adjustment
rate conversion 2-15
frequencies 1-2
normalized 3-51
normalized linear 1-2
terminology 1-2
See also periods
frequency-domain data
displaying 1-100
transforming it into the time domain 4-7

G

generated code
generic real-time (GRT) 2-50
getNumInputs method 8-7
getNumOutputs method 8-7

H

highpass filter designs
continuous-time 3-51
Hz (hertz) 1-2
defined 1-2
See also sample periods

I

IFFT block
using 4-7
importing
arrays 1-61
frame-based signals 1-70
pages of an array 1-61
sample-based matrices 1-61
sample-based signals 1-58
sample-based vector signals 1-58
importing and exporting
frame-based signals 1-70
sample-based signals 1-58
inherit via internal rule 7-26
inheriting sample periods 1-11
initial conditions
with basic algorithmic delay 2-54
input frame periods
defined 2-2
inspecting
frame periods 2-5
frame rates 2-9
sample periods 2-3
sample rates 2-7
inversion of matrices 6-11
isDone method 8-7

L

latency 2-57
due to frame-based processing 1-17
predicting 2-59
reducing 2-57
relation to delay 2-57
libraries
Statistics 6-2
line widths
displaying 2-12
linear algebra
solving linear systems 6-7
locked vs. unlocked mode 8-4
logging
fixed-point data types 7-25
lowpass filter designs
continuous-time 3-51

M

matrices
- factoring 6-9
- format of frame-based 1-15
- inverting 6-11

maximum 6-2
mean 6-2
minimum 6-2
models
- multirate 2-12

modes
- tasking 2-57

modular arithmetic 7-11
multichannel
- frame-based signals 1-38
- sample-based signals 1-32

multichannel signals 1-13
- *See also* signals

multiplication
- fixed-point 7-14

multirate
- blocks 2-58
- demos 3-74
- examples 3-74
- models 2-58

multitasking mode 2-57

N

normalized frequencies 1-2
- defined 1-2
- *See also* frequencies

Nyquist frequency
- defined 1-2

Nyquist rate 1-2

O

output frame periods
- defined 2-2

overflow handling 7-7
overlapping buffers
- causing unintentional rate conversions 2-24

P

padding 7-19
pages of an array
- importing 1-61

parameters
- Buffer overlap, negative values for 2-37
- continuous-time filter 3-51
- FFT length 2-21
- normalized frequency 3-51

Partial Unbuffer block 2-26
partial unbuffering 2-25
passband ripple
- analog filter 3-51

performance
- improving 1-16

periods 1-2
- defined 1-2
- *See also* sample periods *and* frame periods

power spectrum estimation 6-6
precision
- fixed-point data types 7-7

predicting
- tasking latency 2-59

preferences 9-12
preventing unintended rate conversion 2-19
Probe block 2-3
property values 8-5

Q

quantizers
- scalar 5-2
- vector 5-10

R

range
    fixed-point data types 7-7
rate conversion 2-12
    avoiding 2-19
    avoiding rate-mismatch errors 1-7
    blocks for 2-11
    by unbuffering 2-45
    direct 2-11
    frame-rate adjustment 2-12
    frame-size adjustment 2-15
rate types
    block 2-58
    model 2-58
rates 2-2
    auto-promoting 1-9
    See also sample periods *and* frame periods
Real-Time Workshop
    generating generic real-time (GRT) code 2-50
rebuffering 2-25
    altering the sample period of the signal 2-30
    altering the signal 2-26
    causing unintentional rate conversions 2-24
    delay 2-44
    preserving the sample period of the
        signal 2-27
reducing
    latency 2-57
reflection coefficients
    identifying 5-4
reordering channels
    in multichannel frame-based signals 1-54
reset method 8-7
residual signal
    identifying 5-4
ripple
    passband 3-51
rounding
    fixed-point data types 7-8
running
    vector quantization model 5-11
running operations 6-4

S

sample frequency 1-2
    definition 1-2
    See also sample periods
sample modes 2-59
sample periods 1-2
    altered by unbuffering 2-45
    Buffer block 2-26
    continuous-time 1-11
    defined 1-2
    for frame-based signals 2-2
    inherited 1-11
    maintaining constant 2-25
    nonsource blocks 1-11
    of source blocks 1-11
    Rebuffer block 2-26
    related to frame period and frame size 2-2
    Simulink Probe block 2-3
    See also frame periods *and* sample times
sample rates 1-2
    auto-promoting 1-9
    color coding 2-7
    concepts 2-2
    defined 1-2
    inspecting 2-7
    See also sample periods
sample time
    of original time series parameter 2-24
sample times 1-2
    defined 1-2
    in Signal Processing Blockset 1-3
    shifting with sample-time offsets 2-4
    See also sample periods *and* frame periods
sample-based multichannel processing 9-6
sample-based signals 1-13
    combining multichannel signals 1-35

combining single-channel signals 1-32
concatenating multichannel signals 1-35
concatenating single-channel signals 1-32
converting to frame-based 2-33
converting to frame-based with overlap 2-37
creating 1-19
deconstructing multichannel signals 1-42
exporting 1-65
importing 1-58
importing and exporting 1-58
multichannel 1-32
single channel 1-13
splitting multichannel signals 1-42
sample-based single channel processing 9-5
samples
adding 2-25
deleting 2-25
rearranging 2-26
sampling 2-2
See also sample periods *and* frame periods
saturation 7-7
scalar quantizers 5-2
creating 5-5
scaling 7-6
separating
multichannel frame-based signals 1-49
sequences
defining a discrete-time signal 1-2
signals
benefits of frame-based 1-16
characteristics 1-2
continuous-time 1-11
converting frame-based to sample-based 2-45
definition of discrete-time 1-2
definition of frequency 1-2
discrete-time terminology 1-2
frame-based 1-15
inspecting the frame period of 2-5
inspecting the sample period of 2-3
multichannel 1-13
Nyquist frequency 1-2
Nyquist rate 1-2
sample-based 1-13
terminology 1-3
simulations
running from the command line 2-50
single channel signals
frame-based 1-15
sample-based 1-13
single-rate
blocks 2-58
models 2-58
single-tasking mode 2-57
size of a frame 2-11
sliding windows
example 6-3
solvers
fixed-step 1-6
variable-step 1-6
solving
linear systems 6-7
source blocks
defined 1-11
sample periods of 1-11
sources
sample periods of 1-11
speech
analysis and synthesis 5-2
splitting
multichannel frame-based signals into individual signals 1-49
multichannel sample-based signals 1-42
multichannel sample-based signals into individual signals 1-42
multichannel sample-based signals into other multichannel signals 1-45
standard deviation 6-2
statistics
operations 6-2
Statistics library 6-2

step method 8-7
stopband attenuation 3-51
streaming data
  using System objects 8-6
symbols
  time and frequency 1-2
System object
  close method 8-8
  code generation
    signal processing objects 9-3
  creating an instance 8-3
  description 8-2
  fixed point 9-10
    signal processing objects 9-11
  getNumInputs method 8-7
  getNumOutputs method 8-7
  isDone method 8-7
  locked vs. unlocked mode 8-4
  methods 8-6
  preferences 9-12
  properties 8-4
  property values 8-5
  reset method 8-7
  sample- vs. frame-based 9-5
  setting frame-based processing 9-9
  signal processing 9-2
  signal processing example 9-15
  step method 8-7
  tunable property 8-5
  using with Embedded MATLAB 8-9
  value-only input 8-5
system-level settings
  fixed-point 7-25

T

tasking latency 2-57
  example 2-59
  predicting 2-59
tasking modes 2-57
terminology
  sample time and sample period 1-3
  time and frequency 1-2
throughput rates
  increasing 1-17
time-domain data
  displaying 1-79
  transforming it into the frequency domain 4-2
transforming
  frequency-domain data 4-7
  time-domain data 4-2
tunable 8-5
two's complement 7-12

U

unbuffering 2-45
  and rate conversion 2-45
  partial 2-25
  to a sample-based signal 2-26
units of time and frequency measures 1-2
upsampling 2-12
  See also rate conversion
using
  the FFT block 4-2
  the IFFT block 4-7

V

value-only input 8-5
variable-step solver 1-6
vector quantizers 5-10
  configuring the model 5-11
  creating 5-10
  running the model 5-11
viewing
  frequency-domain data 1-100
  time-domain data 1-79

W

wrapping
  fixed-point data types 7-7

Z

zero algorithmic delay 2-51
Zero-Order Hold block 1-11
zero-padding 2-22
  causing unintentional rate conversions 2-24
zeros
  padding with 2-26