Signal Processing Blockset

For Use with Simulink®

- Modeling
- Simulation
- Implementation

User's Guide

The MathWorks

Version 6

How to Contact The MathWorks:

	www.mathworks.com	Web
	comp.soft-sys.matlab	Newsgroup
@	support@mathworks.com	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	Mail

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

Revision History

Contents

Working with Signals

1

i

Advanced Signal Concepts

2

Filters

3

Transforms

4

Working with Fixed-Point Data

8

Blocks — Categorical List

9

Blocks — Alphabetical List

10

Functions — Alphabetical List

11

Glossary

Index

Working with Signals

This chapter helps you understand how sample-based and frame-based signals are represented in Simulink®. 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

Simulink models can process both discrete-time and continuous-time signals. Models built with the Signal Processing Blockset are often intended to process discrete-time signals only. This section defines basic signal terminology and describes how to set the configuration parameters for discrete-time simulations.

This section includes the following topics:

- "Time and Frequency Terminology" on page 1-3 — Review the definitions of common discrete-time signal terminology

- "Recommended Settings for Discrete-Time Simulations" on page 1-5 — Learn the recommended solver algorithms for discrete-time simulations

- "Other Settings for Discrete-Time Simulations" on page 1-7 — Learn the other solver algorithms for discrete-time simulations

Time and Frequency Terminology

A discrete-time signal is a sequence of values that correspond to particular instants in time. The time instants at which the signal is defined are the signal's *sample times*, and the associated signal values are the signal's *samples*. Traditionally, a discrete-time signal is considered to be undefined at points in time between the sample times. For a periodically sampled signal, the equal interval between any pair of consecutive sample times is the signal's *sample period*, T_s. The *sample rate*, F_s, is the reciprocal of the sample period, or $1/T_s$. The sample rate is the number of samples in the signal per second.

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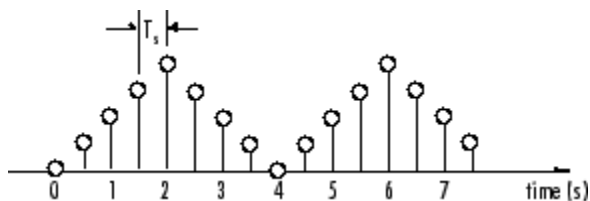
A number of different terms are used to describe the characteristics of discrete-time signals found in Simulink models. These terms, which are listed in the following table, are frequently used to describe the way that various blocks operate on sample-based and frame-based signals.

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

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

Recommended Settings for Discrete-Time Simulations

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

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

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

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

- **Type**: Fixed-step
- **Solver:** discrete (no continuous states)
- **Fixed step size (fundamental sample time)**: auto
- **Tasking mode for periodic sample times**: SingleTasking
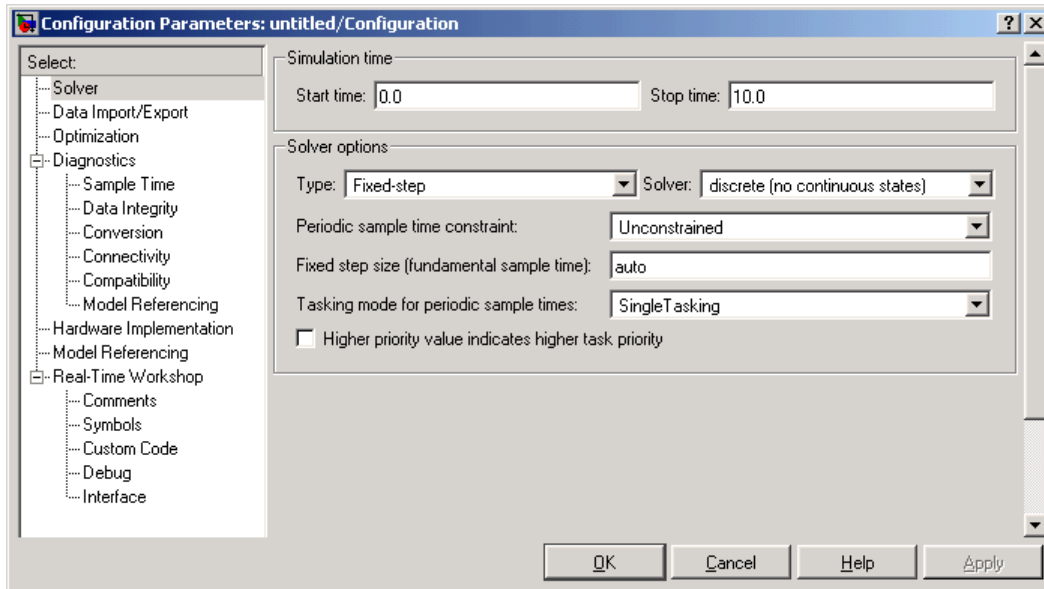
You can automatically set the above solver options for all new models by running the dspstartup M-file. See "Configuring Simulink for Signal Processing Models" in the Getting Started Signal Processing Blockset documentation for more information.

In Fixed-step SingleTasking mode, discrete-time signals differ from the prototype described in "Time and Frequency Terminology" on page 1-3 by remaining defined between sample times. For example, the representation of the discrete-time triangle wave looks like this.
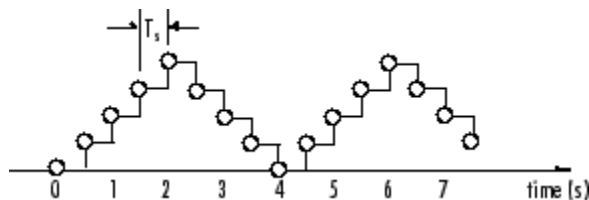
The above signal's value at t=3.112 seconds is the same as the signal's value at t=3 seconds. In Fixed-step SingleTasking mode, a signal's sample times are the instants where the signal is allowed to change values, rather than

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 1-5. 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 2-12 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.
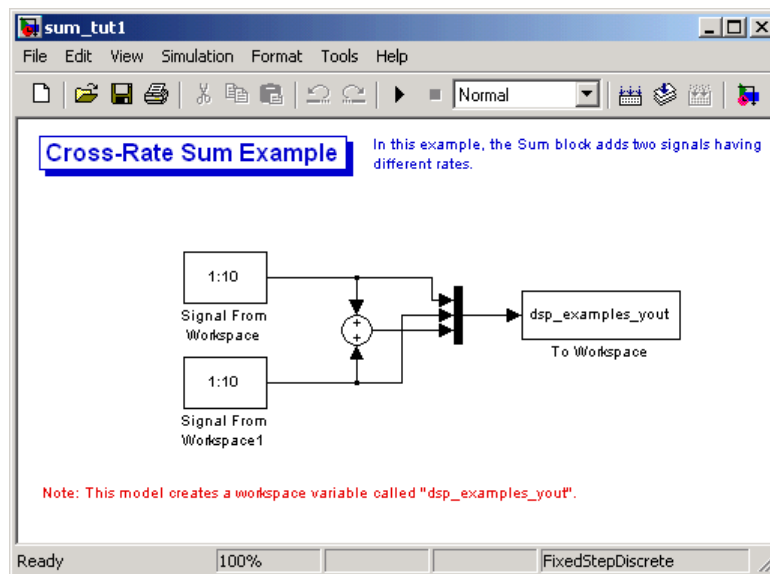
2 Double-click the upper Signal From Workspace block. The **Block Parameters: Signal From Workspace** dialog box opens.

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 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.

---

7 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 implicitly auto-promotes the rate of the

slower signal to match the rate of the faster signal before the addition operation is performed.

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

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.

This section includes the following topics:

- "Continuous-Time Source Blocks" on page 1-11 — Learn how to set up and use continuous-time source blocks
- "Continuous-Time Nonsource Blocks" on page 1-12 — Learn how to use continuous-time nonsource blocks

Continuous-Time Source Blocks

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 block and the Signal Processing Blockset DSP Constant block are examples of continuous-time source blocks. Continuous-time signals are rendered in black when, from the **Format** menu, you point to **Port/Signal Displays** and select **Sample Time 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.
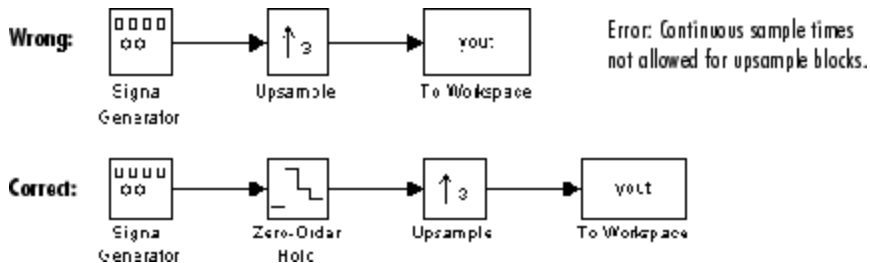
Continuous-Time Nonsource Blocks

Most nonsource blocks in the Signal Processing Blockset 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

Signals can be sample-based or frame-based, single channel or multichannel. The following section discusses sample-based signals in both their single and multichannel form.

This section includes the following topics:

- "Sample-Based Single Channel Signals" on page 1-13 — Learn about the characteristics of a sample-based single channel signal
- "Sample-Based Multichannel Signals" on page 1-13 — Learn about the characteristics of a sample-based multichannel signal

Sample-Based Single Channel Signals

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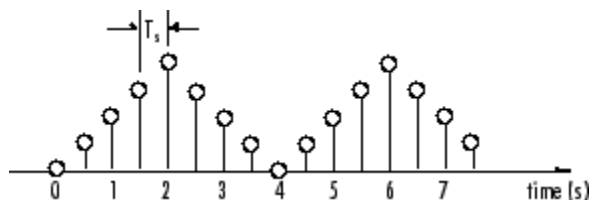
The representation of single-channel signals is actually a special case of the general multichannel signal.

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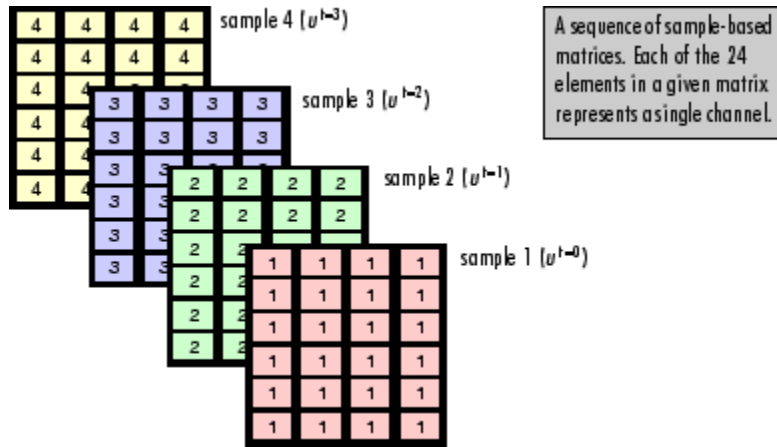
The signal in channel 1 is composed of the following sequence:

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

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

Signals can be sample-based or frame-based, single channel or multichannel. The following section discusses frame-based signals in both their single and multichannel form. It also explains how frame-based processing accelerates real-time systems and simulations.

This section contains the following topics:

- "Frame-Based Single Channel Signals" on page 1-15 — Learn about the characteristics of a frame-based single channel signal

- "Frame-Based Multichannel Signals" on page 1-15 — Learn about the characteristics of a frame-based multichannel signal

- "Benefits of Frame-Based Processing" on page 1-17 — Understand how frame-based processing accelerates real-time systems and simulations

Frame-Based Single Channel Signals

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 single-channel signal, because there is only one independent sequence of numbers.
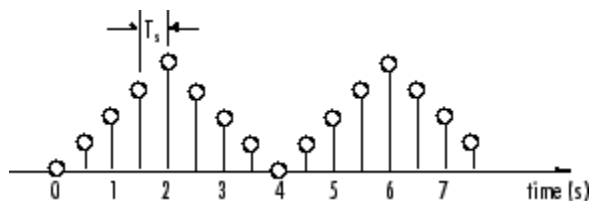
Frame-based single channel signals are represented as vectors. An M-by-1 frame-based vector represents M consecutive samples from a single channel. In other words, each matrix row represents one sample, or time slice, from one distinct channel.

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.

For example, this 6-by-4 matrix represents a four-channel frame-based signal with six samples per frame.

Frame matrix:
4 channels,
1 frame per channel,
6 samples per frame

ch1 ch2 ch3 ch4
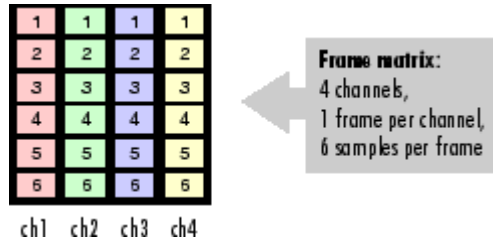
Consider a sequence of frame matrices, where $u^{t=0}$ is the first matrix in a series, $u^{t=1}$ is the second, $u^{t=2}$ is the third, and so on.

A sequence of frame-based matrices. Each column in a given matrix represents a single channel.
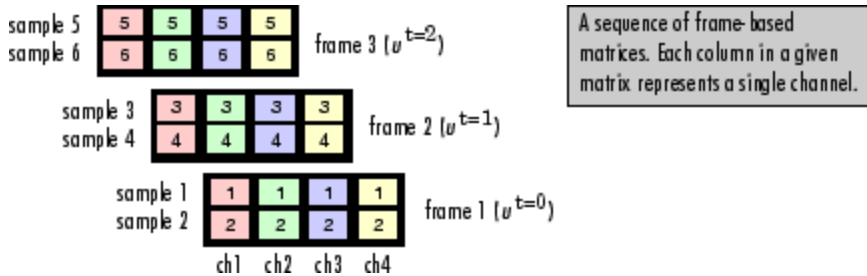
The signal in channel 1 is the following sequence:

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

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.
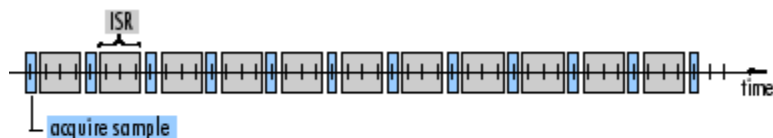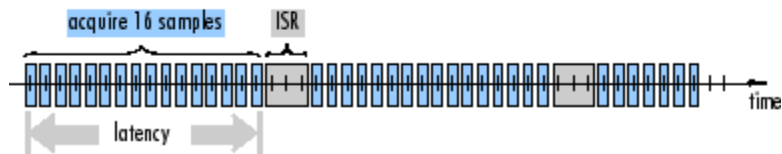
It's important to note that frame-based processing introduces a certain amount of latency into a process due to the inherent lag in buffering the initial frame. In many instances, however, it is possible to select frame sizes that improve throughput without creating unacceptable latencies. For more information, see "Delay and Latency" on page 2-48.

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

A sample-based signal is propagated through a model one sample at a time. This section describes two ways to create a sample-based signal.

This section includes the following topics:

- "Using the DSP Constant Block" on page 1-19 — Create a six-channel, constant sample-based signal using the DSP Constant block
- "Using the Signal from Workspace Block" on page 1-22 — Create a four-channel sample-based signal using the Signal From Workspace block

Using the DSP 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
- DSP Constant
- Identity Matrix

The most versatile of the blocks listed above is the DSP Constant block. This topic discusses how to create a constant sample-based signal using the DSP Constant block:

1 Create a new Simulink model.

2 From the Signal Processing Sources library, click-and-drag a DSP 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 DSP Constant block, and set the block parameters as follows:

- **Constant value** = [1 2 3; 4 5 6]

- **Sample mode** = `Discrete`
- **Output** = `Sample-based`
- **Sample time** = `1`

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

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

The model should now look similar to the following figure. You can also open the model by typing `doc_usingdspcnstblksb` at the MATLAB command line.
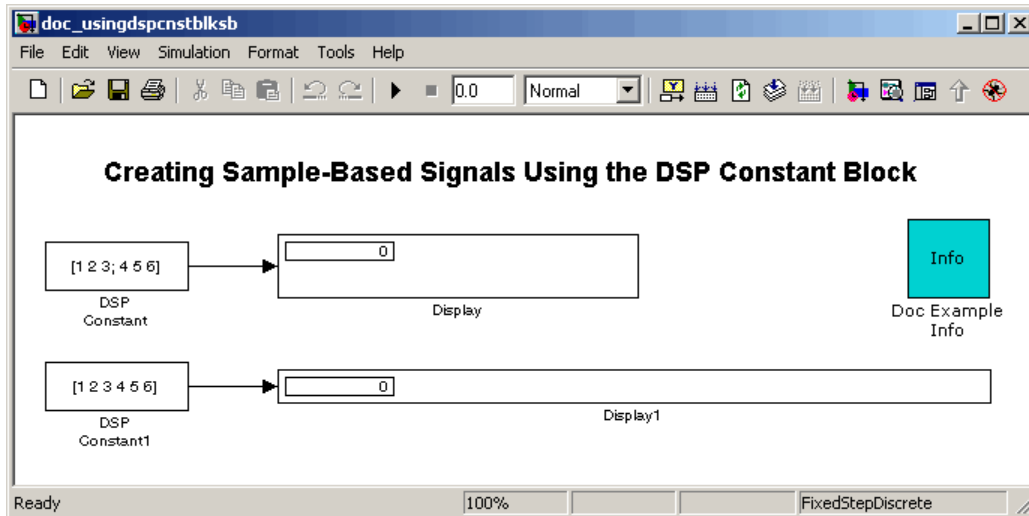
You have now successfully created a six-channel, constant sample-based signal with a sample period of 1 second.

Creating a 1-D Vector Signal

You can modify the previous model in order to create a 1-D vector signal:

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

- **Constant value** = [1 2 3 4 5 6]

- **Output** = Sample-based (interpret vector as 1-D)

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

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

The following figure shows the results of these two procedures.
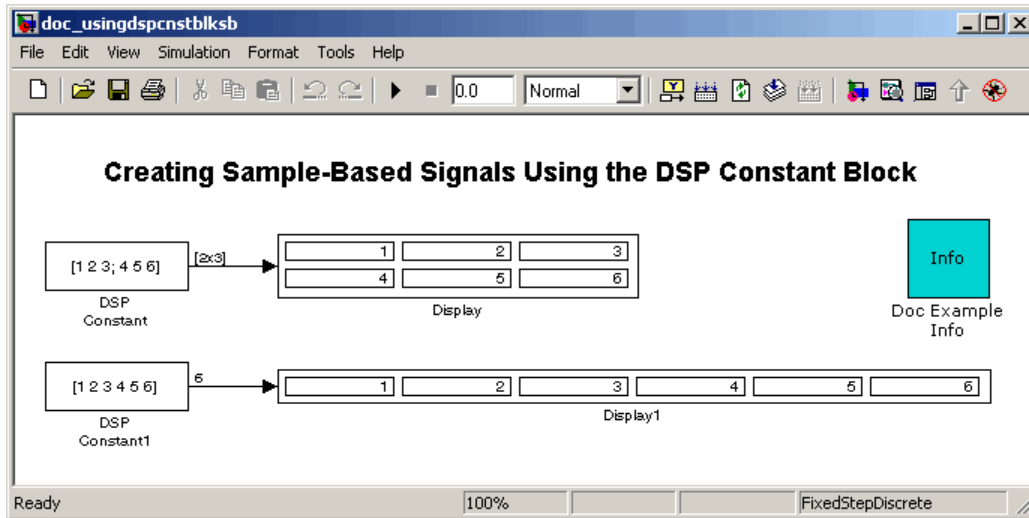
The DSP Constant block generates a length-6 1-D vector signal. This means that the output is not a matrix. However, most nonsource signal processing blocks interpret a length-M 1-D vector as an M-by-1 matrix (column vector).

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.
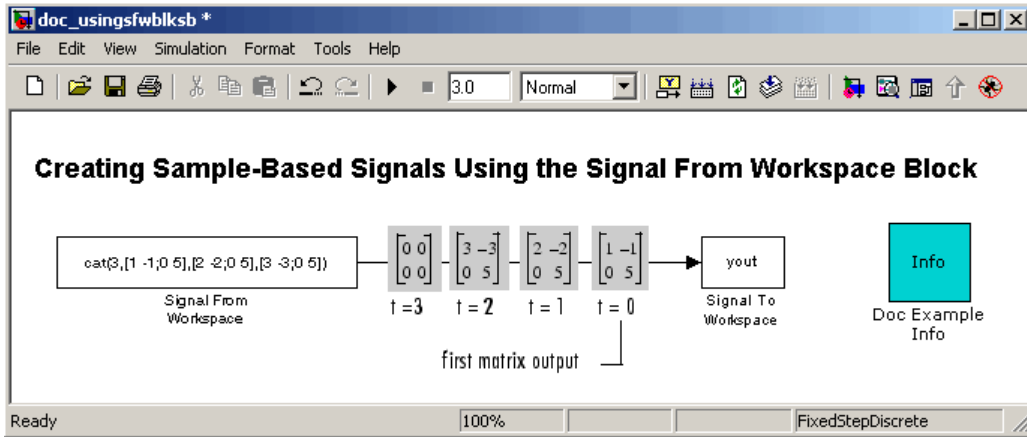
9 At the MATLAB command line, type yout.

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

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 section describes two ways to create frame-based signals.

This section includes the following topics:

- "Using the Sine Wave Block" on page 1-25 — Create a three-channel frame-based signal using the Sine Wave block
- "Using the Signal from Workspace Block" on page 1-28 — Create a two-channel frame-based signal using the Signal From Workspace block

Using the Sine Wave Block

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

- Chirp
- Discrete Impulse
- 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, and set the **Sum along** parameter to Rows. 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.
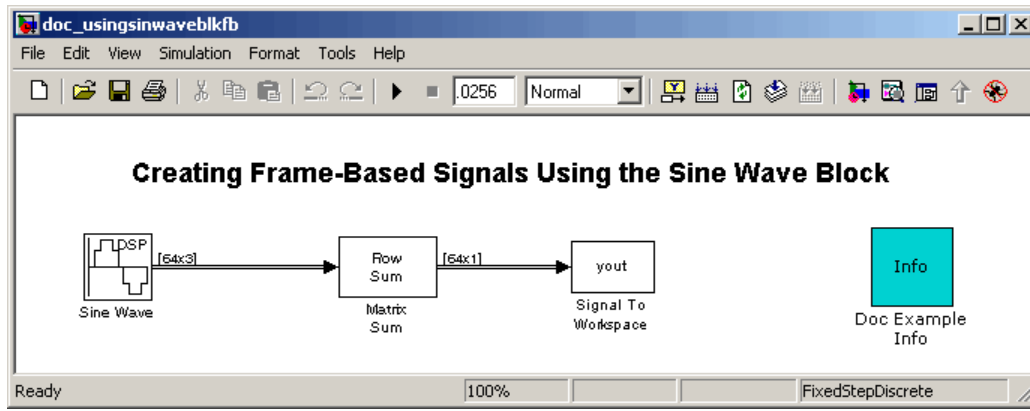
The three signals are summed point-by-point by a Matrix Sum block. Then, they are exported to the MATLAB workspace.

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

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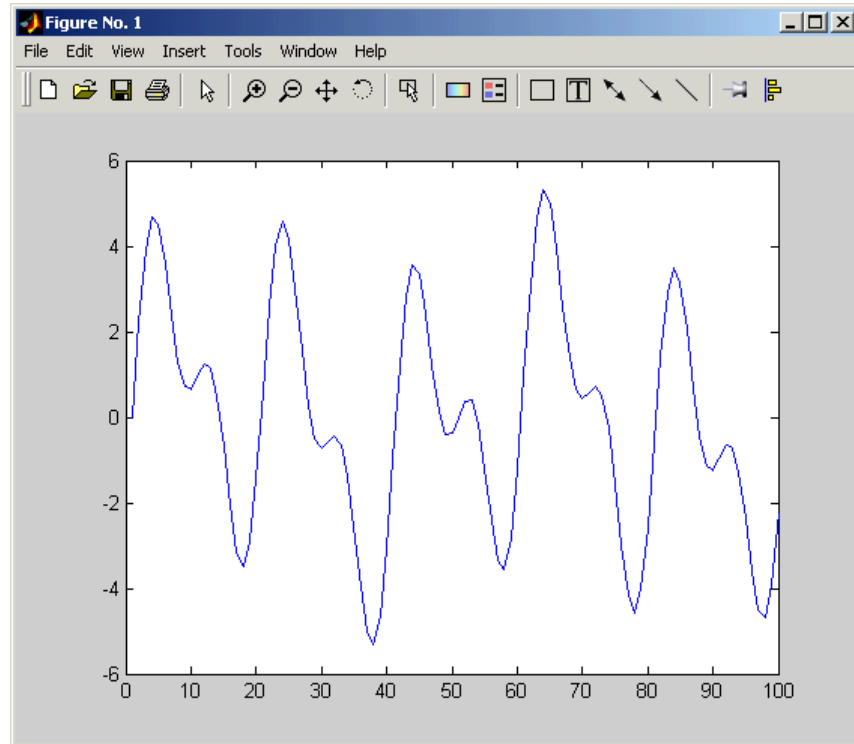
This figure represents a portion of the sum of the three sinusoids. You have now added the channels of a three-channel frame-based signal together and displayed the results in a figure window.

Using the Signal from Workspace Block

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.
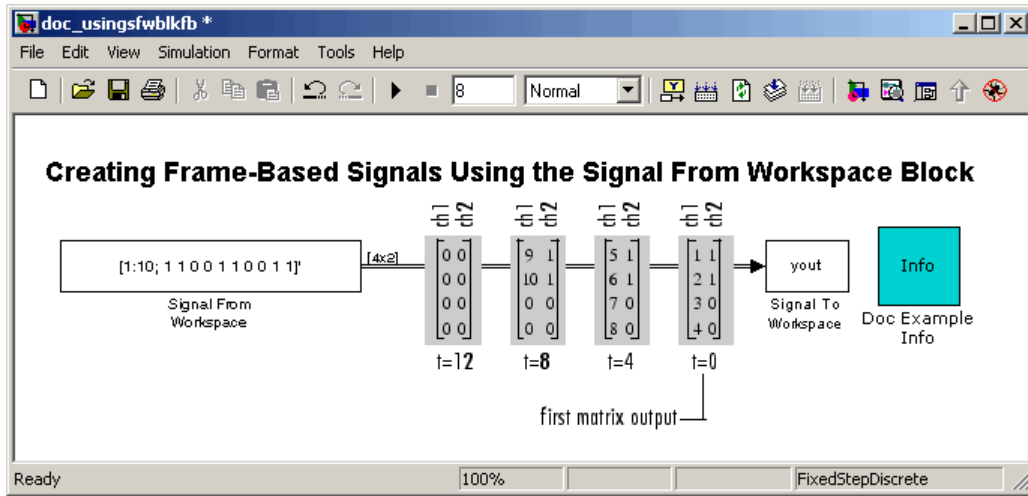
9 At the MATLAB command line, type yout.

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

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 Matrix Concatenation block. In addition, several multichannel sample-based signals can be combined into a single multichannel sample-based signal using the same technique.

This section contains the following topics:

- "Combining Single-Channel Sample-Based Signals" on page 1-31 — Create a multichannel sample-based signal from several individual sample-based signals

- "Combining Multichannel Sample-Based Signals" on page 1-34 — Create a multichannel sample-based signal from several multichannel sample-based signals

Combining Single-Channel Sample-Based Signals

You can combine individual sample-based signals into a multichannel signal by using the Matrix Concatenation block in the Simulink Math Operations library:

1 Open the Matrix Concatenation Example 1 model by typing

```
doc_cmbsnglchsbsigs
```

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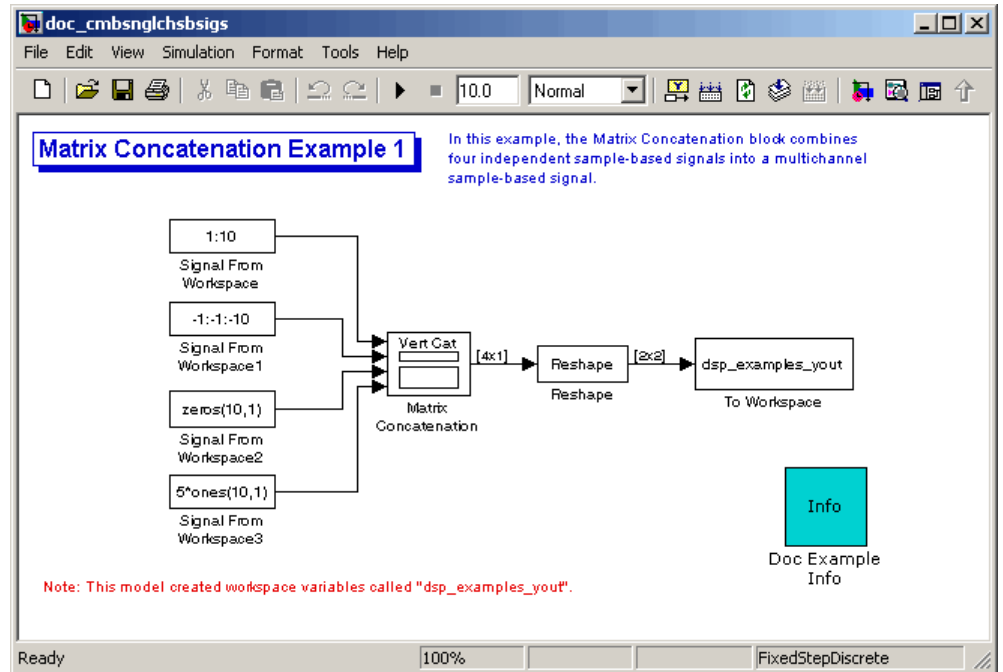
2 Double-click the Signal From Workspace block, and set the **Signal** parameter to 1:10. Click **OK**.

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

- **Number of inputs** = 4
- **Concatenation method** = Vertical

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.
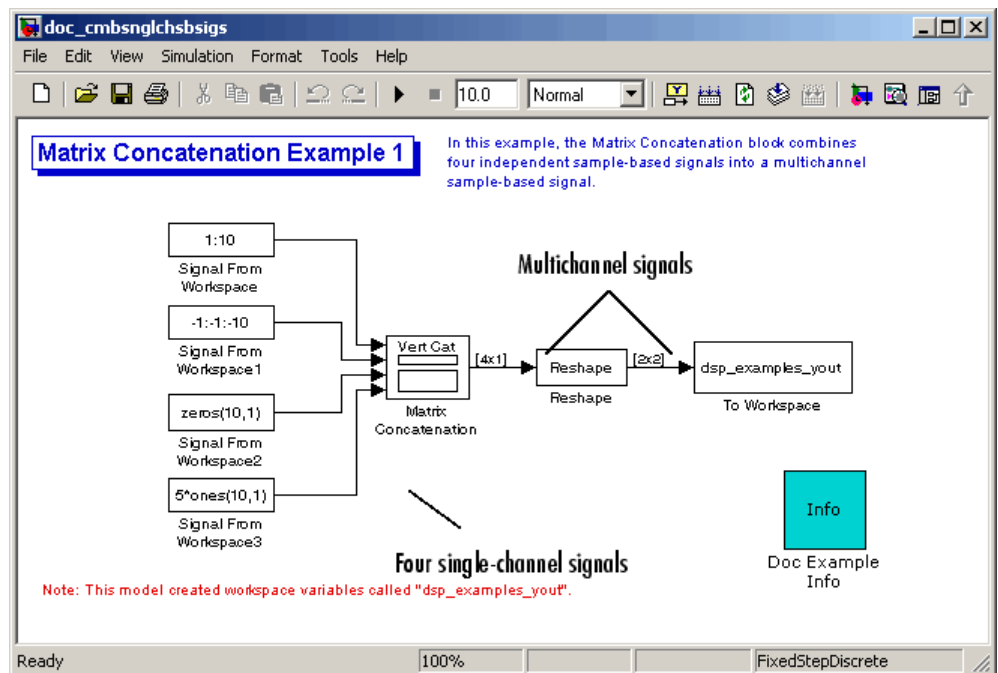
Each 4-by-1 output from the Matrix Concatenation block contains one sample from each of the four input signals at the same instant in time. The Reshape block rearranges the samples into a 2-by-2 matrix. Each element of this matrix is a separate channel. Note that the Reshape block works columnwise, so that a column vector input is reshaped as shown below.
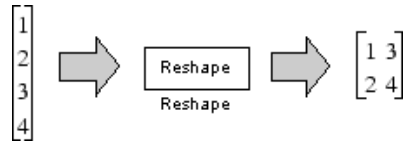
The 4-by-1 matrix output by the Matrix Concatenation block and the 2-by-2 matrix output by the Reshape block in the above model represent the same four-channel sample-based signal. In some cases, one representation of the signal may be more useful than the other.

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 Concatenation block:

1 Open the Matrix Concatenation Example 2 model by typing

    doc_cmbmltichsbsigs

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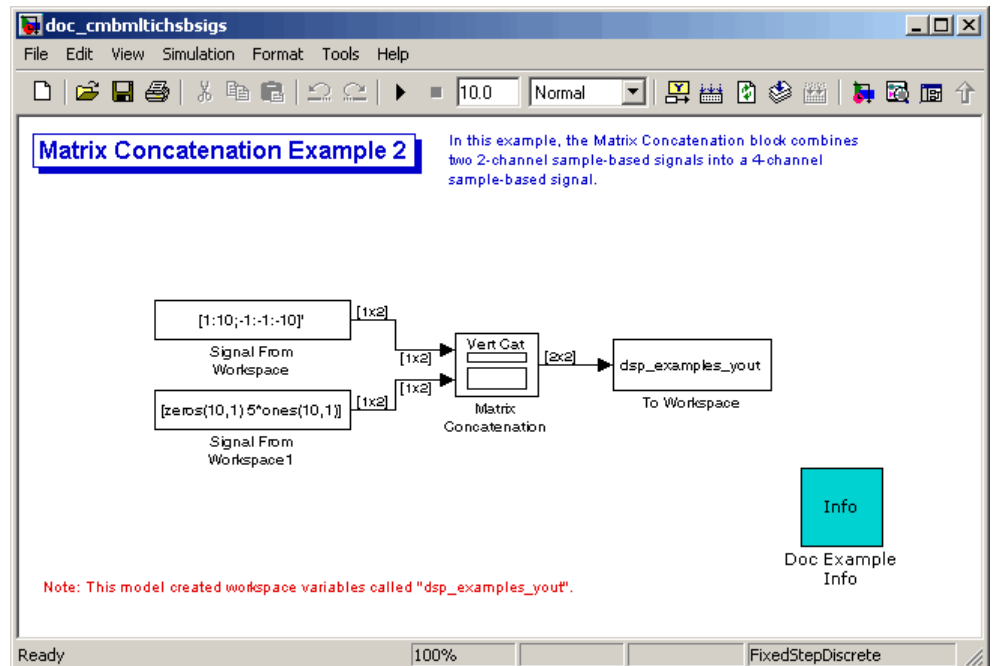
2 Double-click the Signal From Workspace block, and set the **Signal** parameter to [1:10;-1:-1:-10]'. Click **OK**.

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

   • **Number of inputs** = 2

   • **Concatenation method** = Vertical

5 Run the model.

   The model combines both two-channel sample-based signals into a four-channel signal.
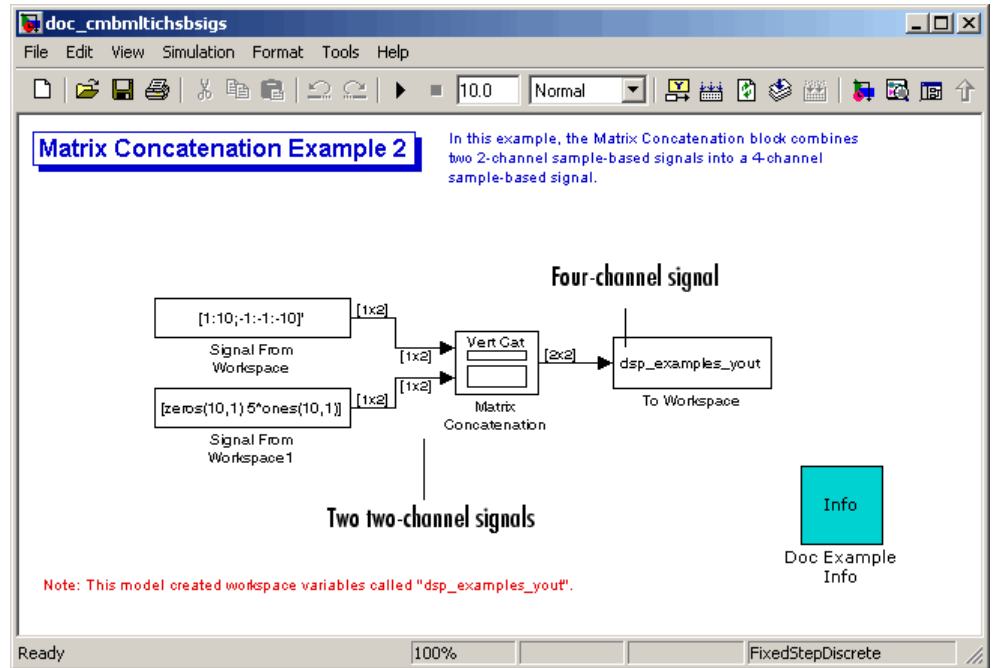
Each 2-by-2 output from the Matrix Concatenation block contains both samples from each of the two input signals at the same instant in time. Each element of this matrix is a separate channel.

Creating Multichannel Frame-Based Signals

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 Concatenation block. Individual signals can be added to an existing multichannel signal in the same way.
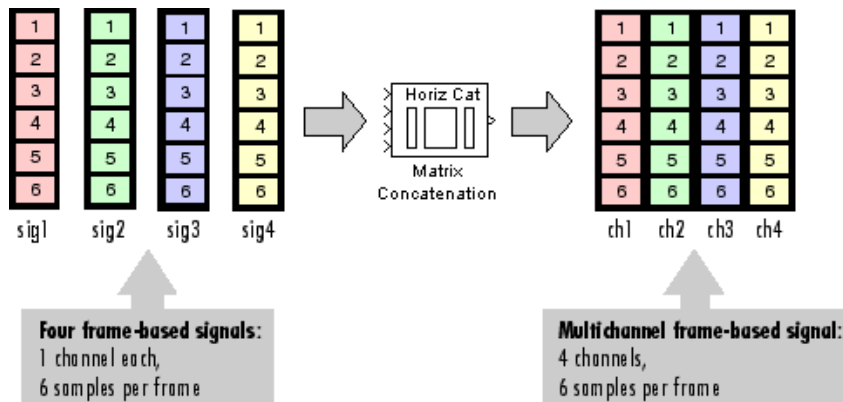
This section contains the following topic:

- "Combining Frame-Based Signals" on page 1-37 — Create a multichannel frame-based signal from several individual frame-based signals

Combining Frame-Based Signals

You can combine existing frame-based signals into a larger multichannel signal by using the Simulink Matrix Concatenation 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 Concatenation Example 3 model by typing

```
doc_combiningfbsigs
```

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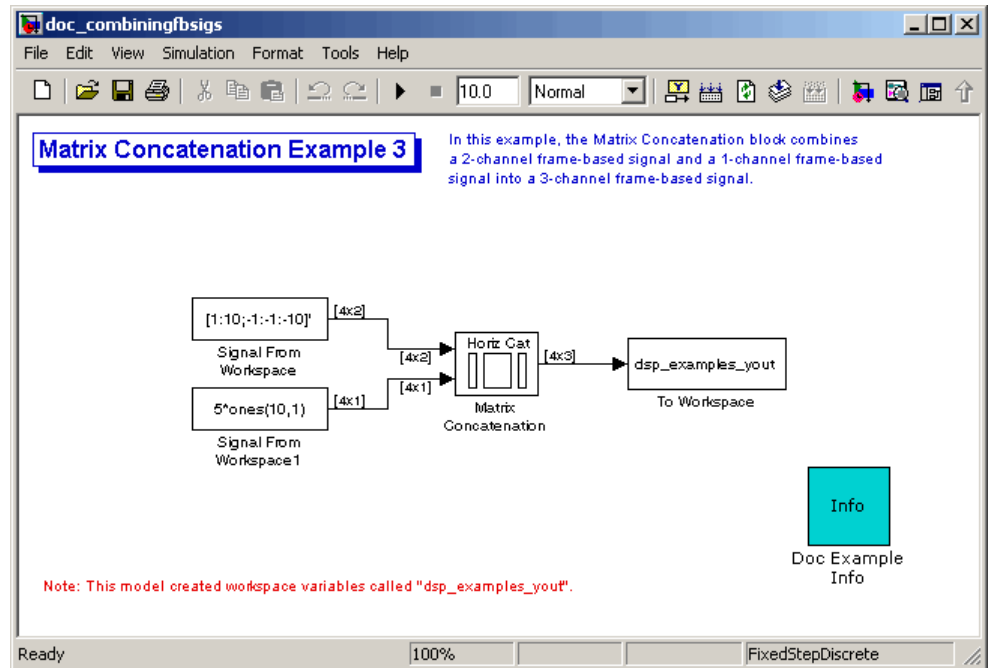
2 Double-click the Signal From Workspace block. Set the block parameters as follows:

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

- **Number of inputs** = `2`
- **Concatenation method** = `Horizontal`

6 Run the model.

The figure below is a graphical representation of what happens to one input frame during simulation.
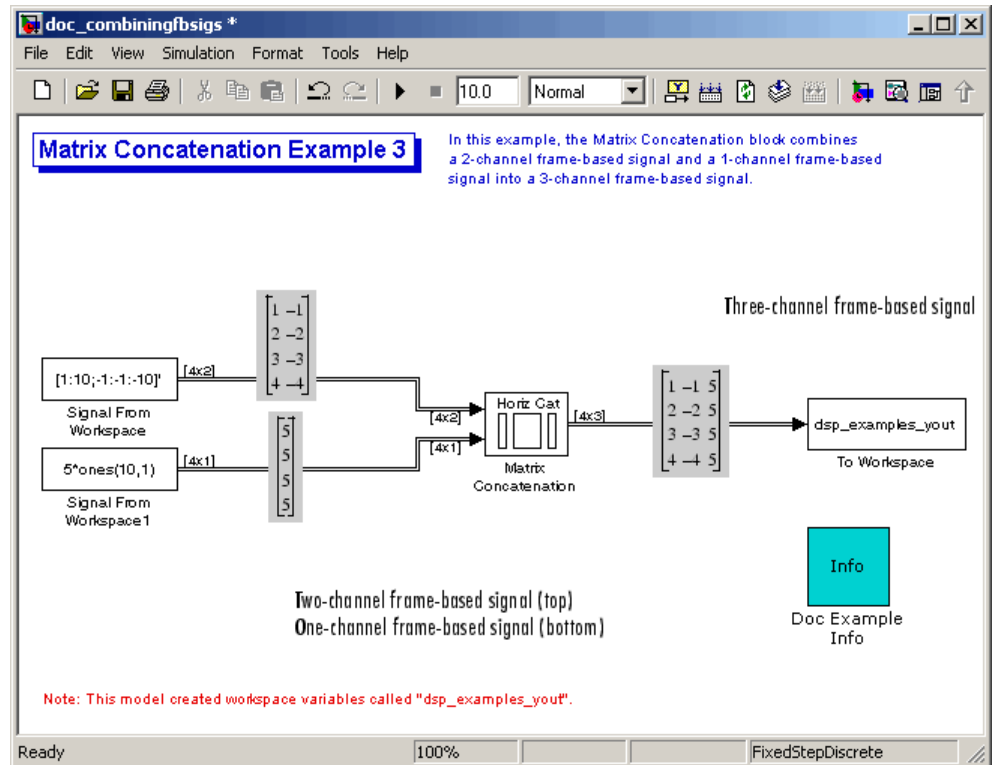
The 4-by-3 matrix output from the Matrix Concatenation block contains all three input channels, and preserves their common frame rate and frame size.

Deconstructing Multichannel Sample-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.

This section includes the following topics:

- "Splitting Multichannel Sample-Based Signals into Individual Signals" on page 1-41 — Use the Multiport Selector block to create three, single-channel sample-based signals from a multichannel sample-based signal

- "Splitting Multichannel Sample-Based Signals into Several Multichannel Signals" on page 1-43 — Use the Submatrix block to create a six-channel sample-based signal from a 35-channel sample-based signal.

Splitting Multichannel Sample-Based Signals into Individual Signals

You can split multichannel sample-based signal into single-channel sample-based signals using the Multiport Selector block. This blocks allows you to select specific rows and/or columns and propagate this 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.
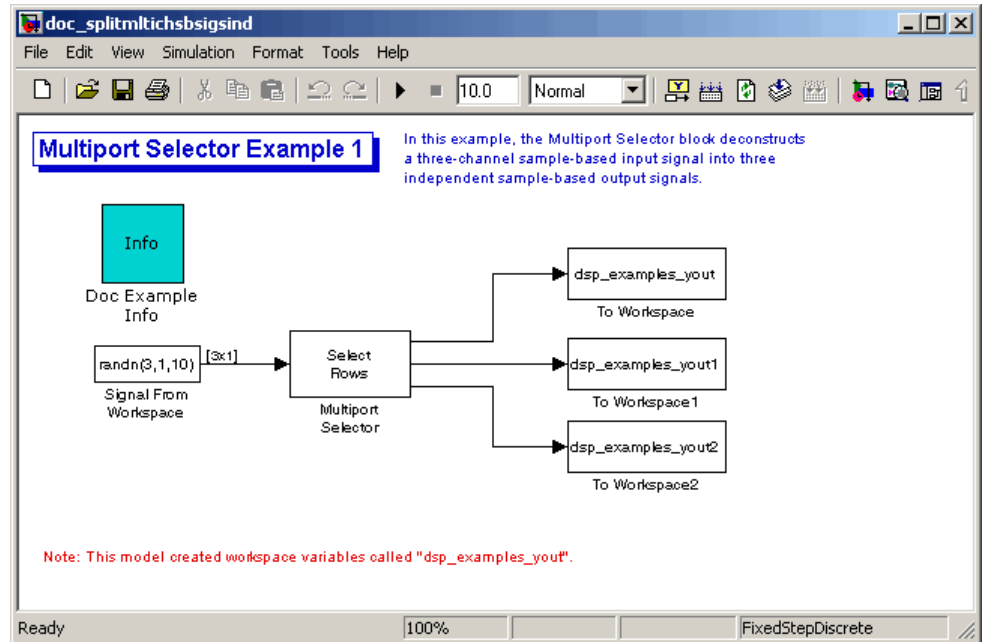
2 Double-click the Signal From Workspace block, and set the block parameters as follows:

- **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

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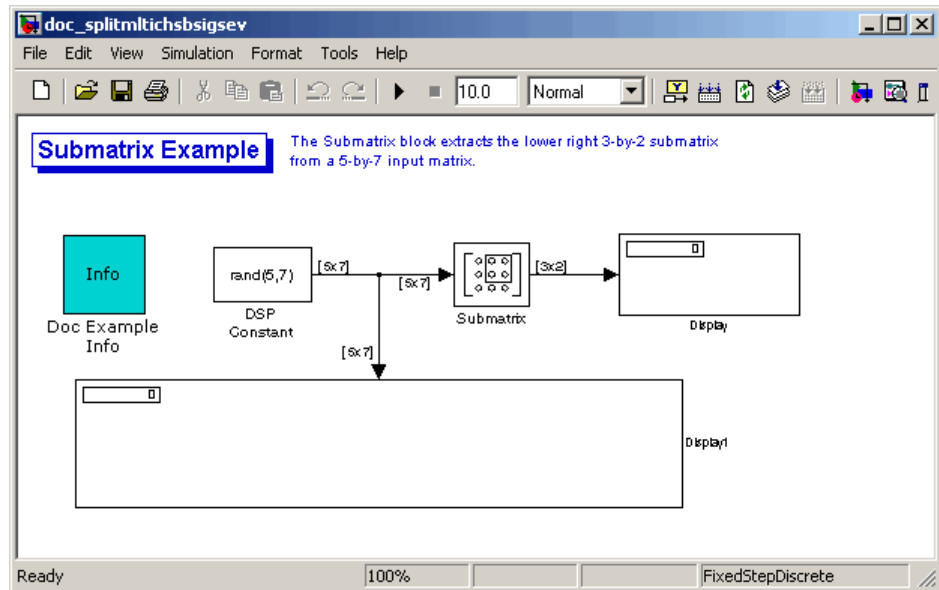
2 Double-click the DSP Constant block, and set the block parameters as follows:

- **Constant value** = rand(5,7)

- **Output** = Sample-based

Based on these parameters, the DSP 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 index** = 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.
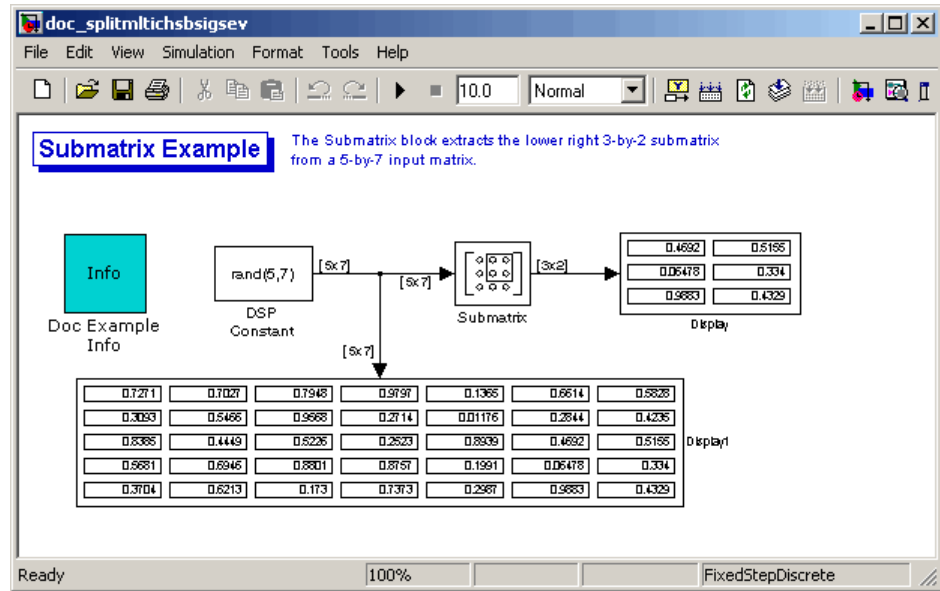
Notice that the output of the Submatrix block is equivalent to the matrix created by rows three through five and columns six through seven of the input matrix.

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

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.

This section includes the following topics:

- "Splitting Multichannel Frame-Based Signals into Individual Signals" on page 1-47 — Use the Multiport Selector block to create a single-channel and a two-channel frame-based signal from a multichannel frame-based signal

- "Reordering Channels in Multichannel Frame-Based Signals" on page 1-51 — Use the Permute Matrix block to rearrange the channels in a frame-based signal

Splitting Multichannel Frame-Based Signals into Individual Signals

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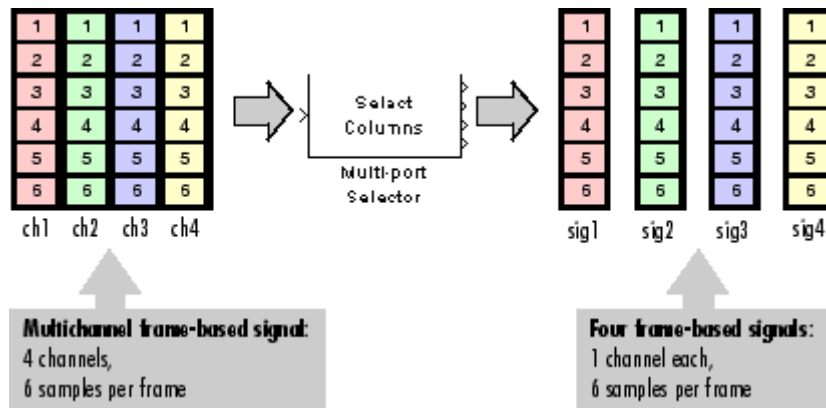
In this example, you use the Multiport Selector block to extract a single-channel and a two channel frame-based signal from a multichannel frame-based signal:

1 Open the Multiport Selector Example 2 model by typing

    doc_splitmltichfbsigsind

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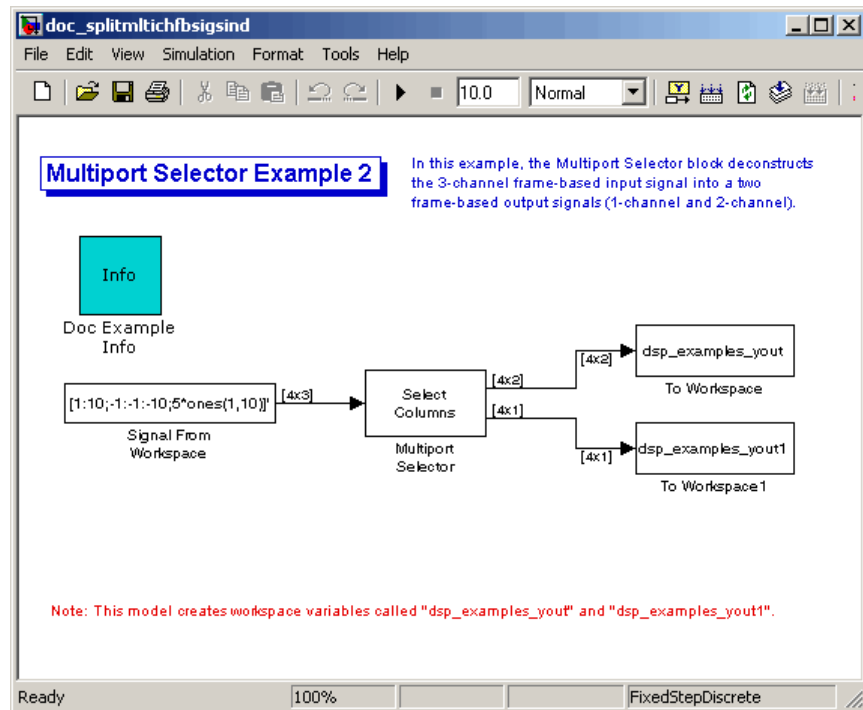
2 Double-click the Signal From Workspace block, and set the block parameters as follows:

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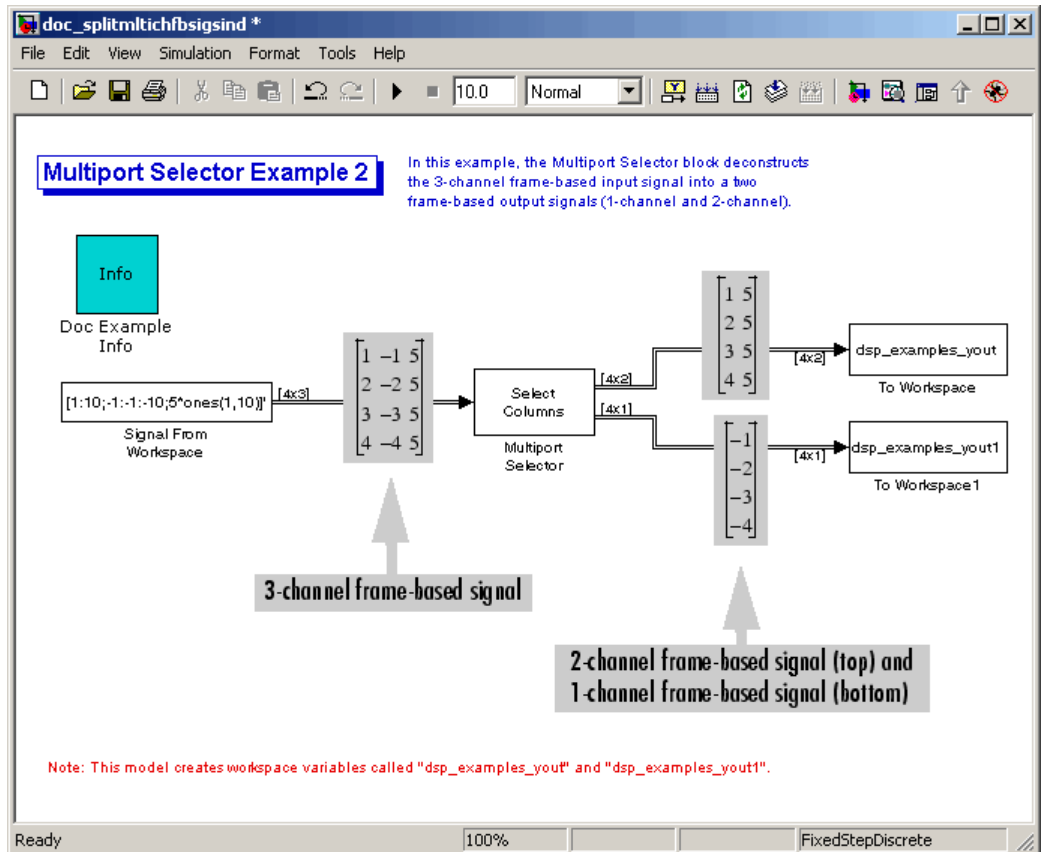
The Multiport Selector block outputs a two-channel frame-based signal, comprised of the first and third column of the input signal, at the first port. It outputs a single-channel frame-based signal, comprised of the second column of the input signal, at the second port.

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

Some blocks in the Signal Processing Blockset 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.
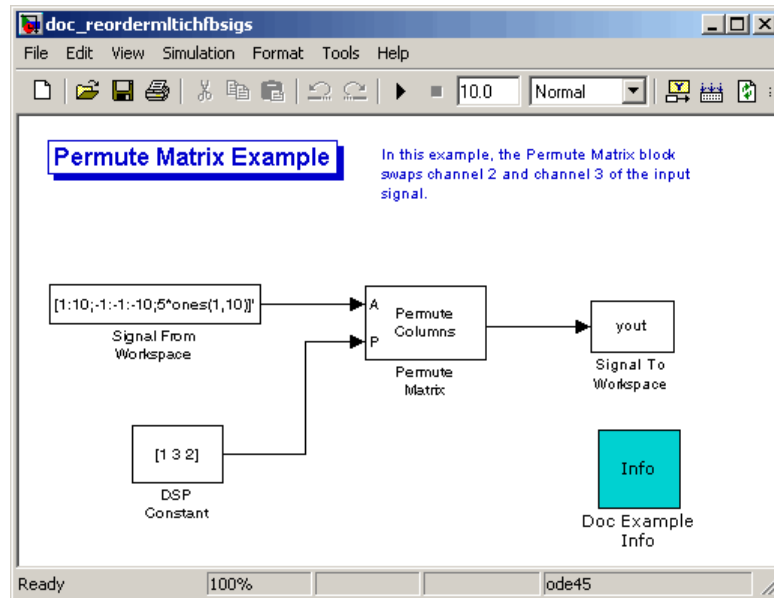
2 Double-click the Signal From Workspace block, and set the block
parameters as follows:

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

- **Constant value** = [1 3 2]

- **Sample mode** = Discrete

- **Output** = Frame-based

- **Frame period** = 4

The discrete-time, frame-based vector output by the DSP 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 DSP 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.

1-53
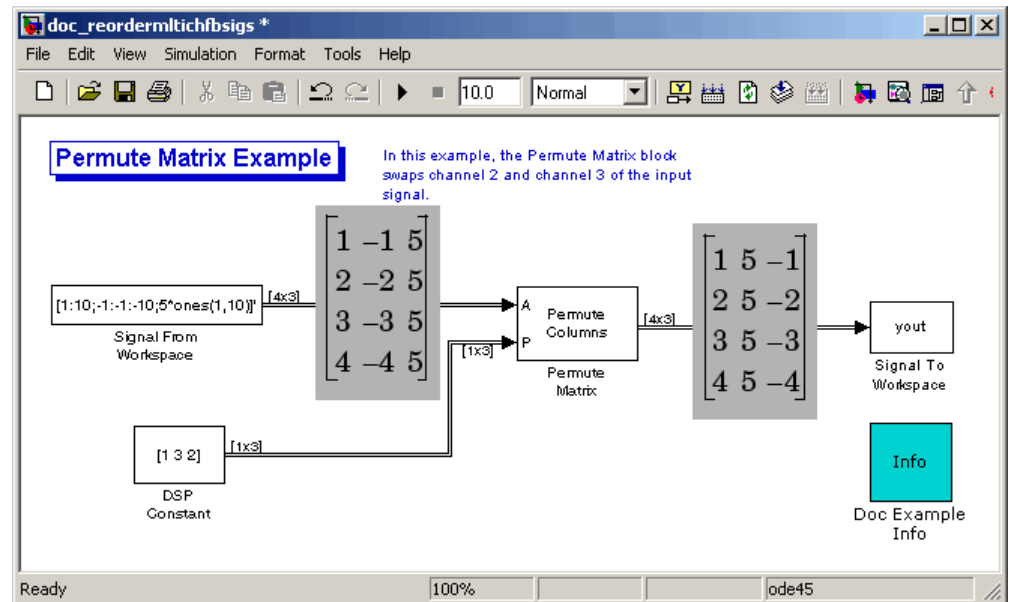
The second and third channel of the frame-based input signal are swapped.

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

Although a number of signal generation blocks are available in both the Simulink and the Signal Processing Blockset libraries, it is also possible to import custom signals from the MATLAB workspace into your Simulink model. The Signal From Workspace block in the Signal Processing Sources library is the key block for importing sample-based signals of all dimensions from the MATLAB workspace. The Signal To Workspace block in the Signal Processing Sinks library can be used to export sample-based signals to the MATLAB workspace

This section includes the following topics:

- "Importing Sample-Based Vector Signals" on page 1-55 — Use the Signal From Workspace block to import a sample-based vector signal into your signal processing model

- "Importing Sample-Based Matrix Signals" on page 1-58 — Use the Signal From Workspace block to import a sample-based matrix signal into your signal processing model

- "Exporting Sample-Based Signals" on page 1-62 — Use the Signal To Workspace block to export a sample-based matrix signal to your MATLAB workspace

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.
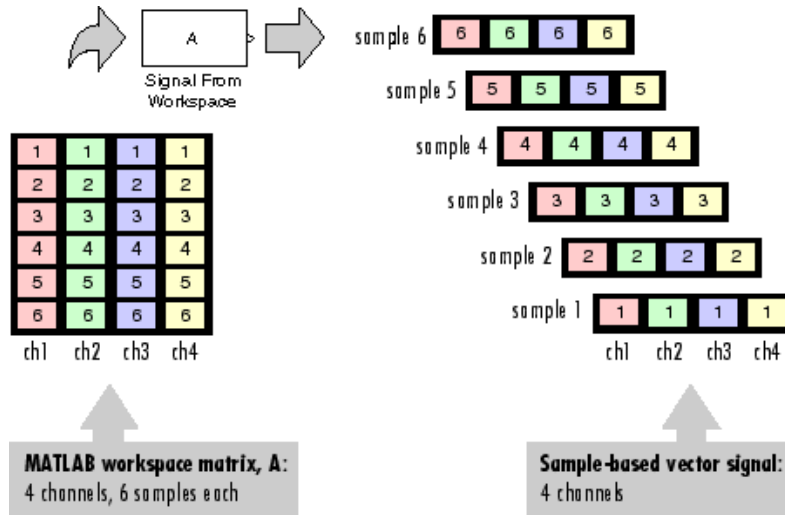
In the following example, you use the Signal From Workspace block to import a sample-based vector signal into your model:

1 Open the Signal From Workspace Example 3 model by typing

    doc_importsbvectorsigs

at the MATLAB command line.
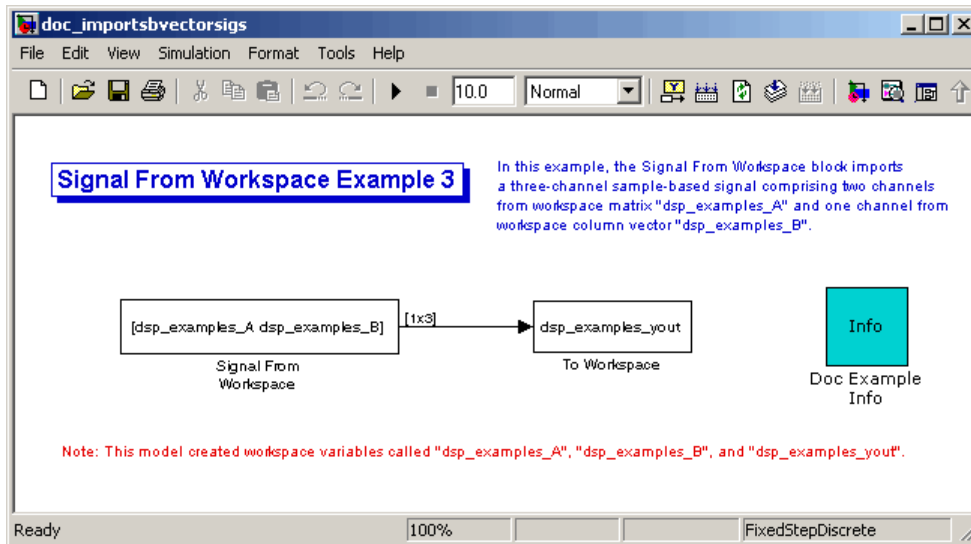
2 At the MATLAB command line, type A = [1:100;-1:-1:-100]';

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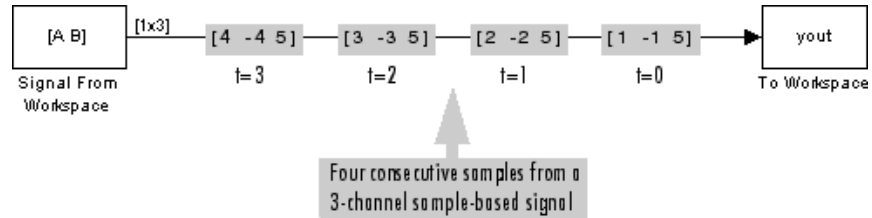
The first row of the input matrix [A B] is output at time t=0, the second row of the input matrix is output at time t=1, and so on.

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.
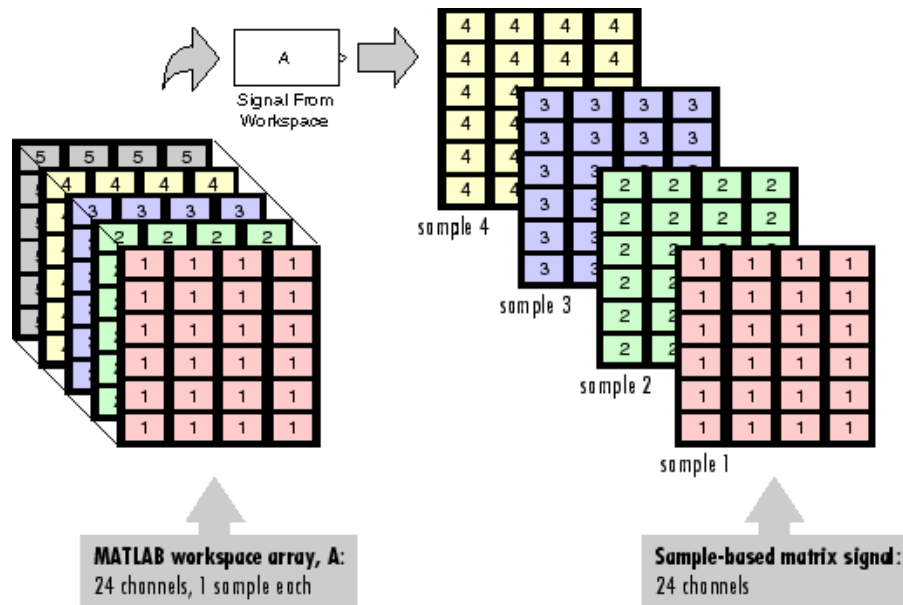
In the following example, you use the Signal From Workspace block to import a four-channel, sample-based matrix signal into a Simulink model:

1 Open the Signal From Workspace Example 4 model by typing

```
doc_importsbmatrixsigs
```

at the MATLAB command line.
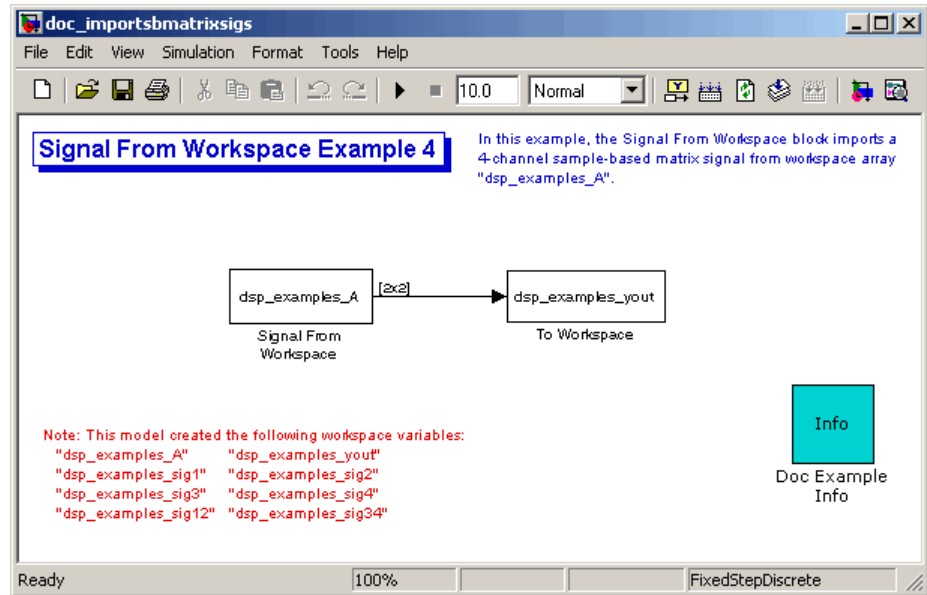
Also, the following variables are loaded into the MATLAB workspace:

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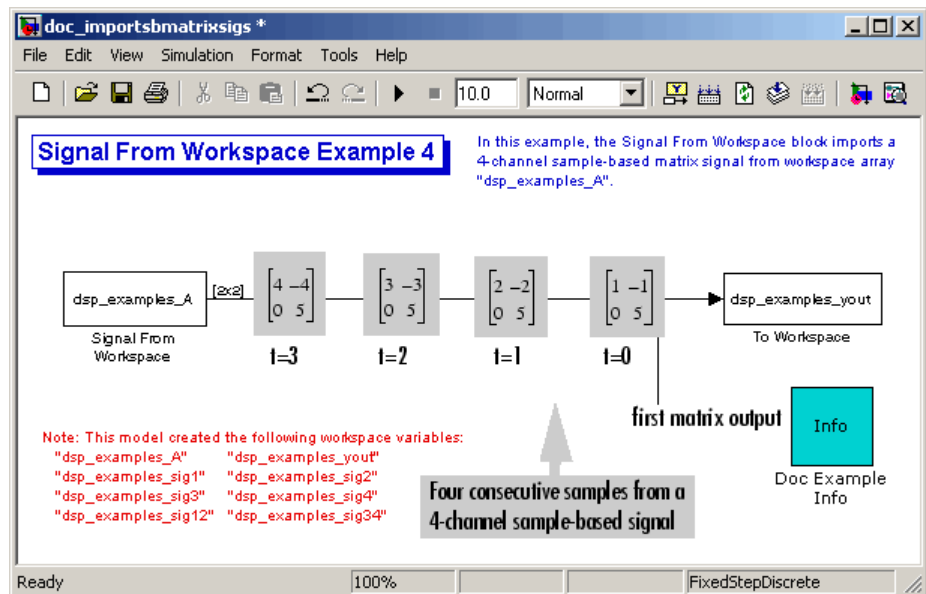
The Signal From Workspace block imports the four-channel sample based signal from the MATLAB workspace into the Simulink model one matrix at a time.

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 back of the array.

The figure below is the graphical illustration of this process using a 6-by-4 sample-based signal exported to workspace array A.
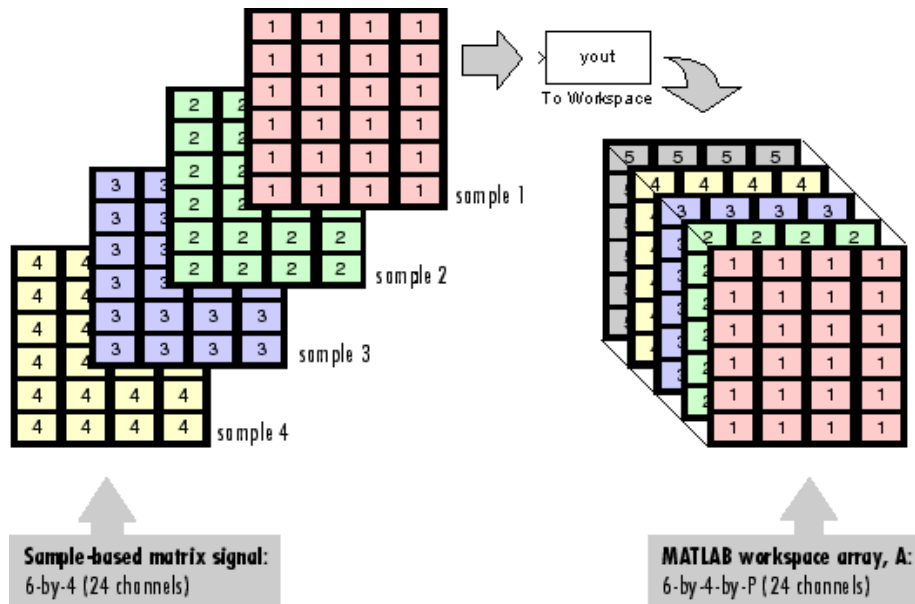
The workspace array always has time running along its third dimension, P. Samples are saved along the P dimension whether the input is a matrix, vector, or scalar (single channel case).

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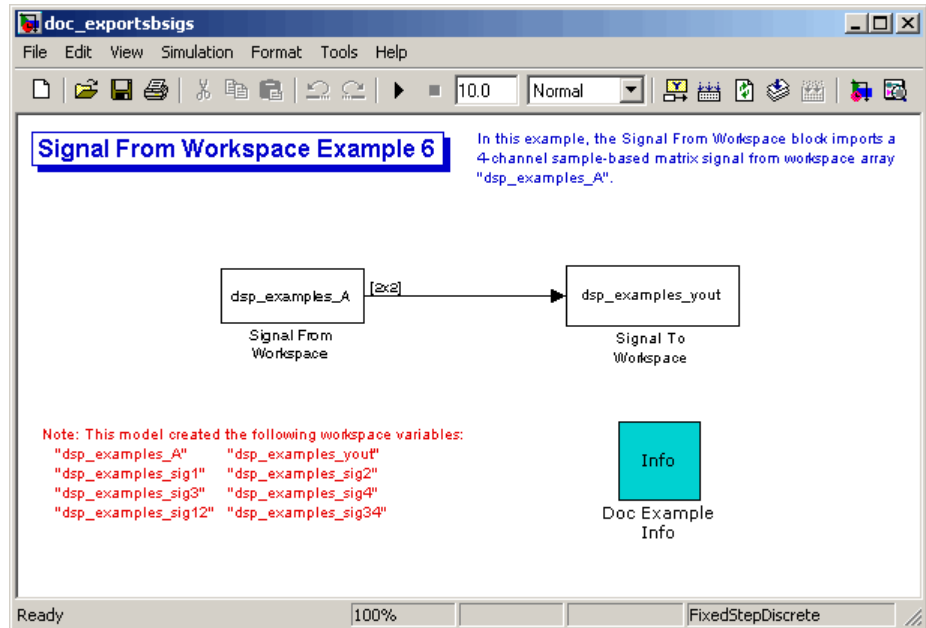
Also, the following variables are loaded into the MATLAB workspace:

```
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
```

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

Although a number of signal generation blocks are available in both the Simulink and the Signal Processing Blockset libraries, it is also possible to import frame-based signals from the MATLAB workspace into your Simulink model. The Signal From Workspace block in the Signal Processing Sources library is the key block for importing frame-based signals of all dimensions from the MATLAB workspace. The Signal To Workspace block in the Signal Processing Sinks library can be used to export frame-based signals to the MATLAB workspace

This section includes the following topics:

- "Importing Frame-Based Signals" on page 1-66 — Use the Signal From Workspace block to create a three-channel, frame-based signal and import it into your model.

- "Exporting Frame-Based Signals" on page 1-69 — Use the Signal To Workspace block to export a three-channel, frame-based signal into the MATLAB workspace.

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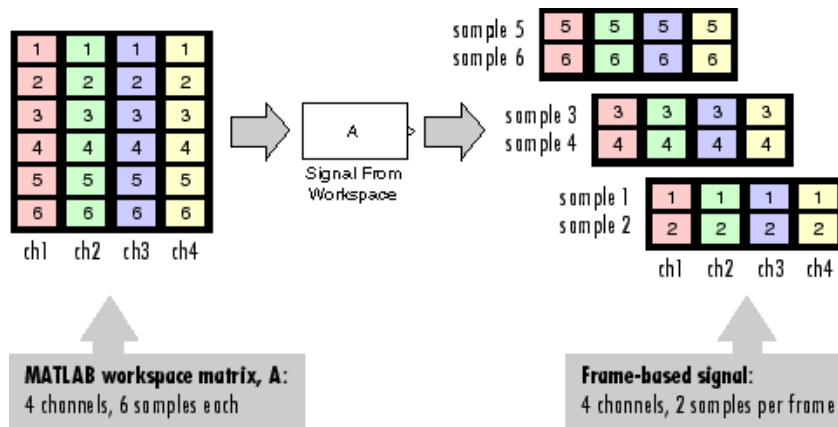
Note Although independent channels are generally represented as columns, a single-channel signal can be represented in the workspace as either a column vector or row vector. The output from the Signal From Workspace block is a column vector in both cases.

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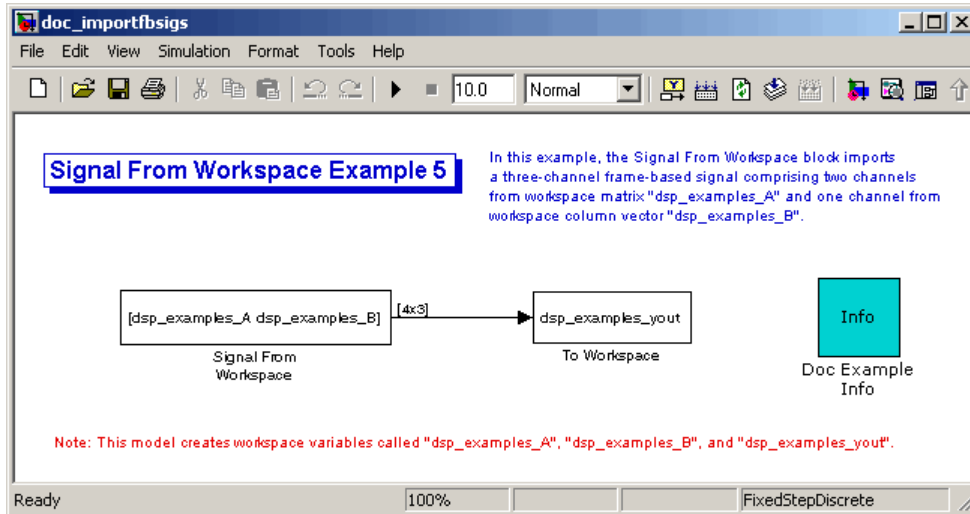
2 Double-click the Signal From Workspace block. Set the block parameters as follows, and then click **OK**:

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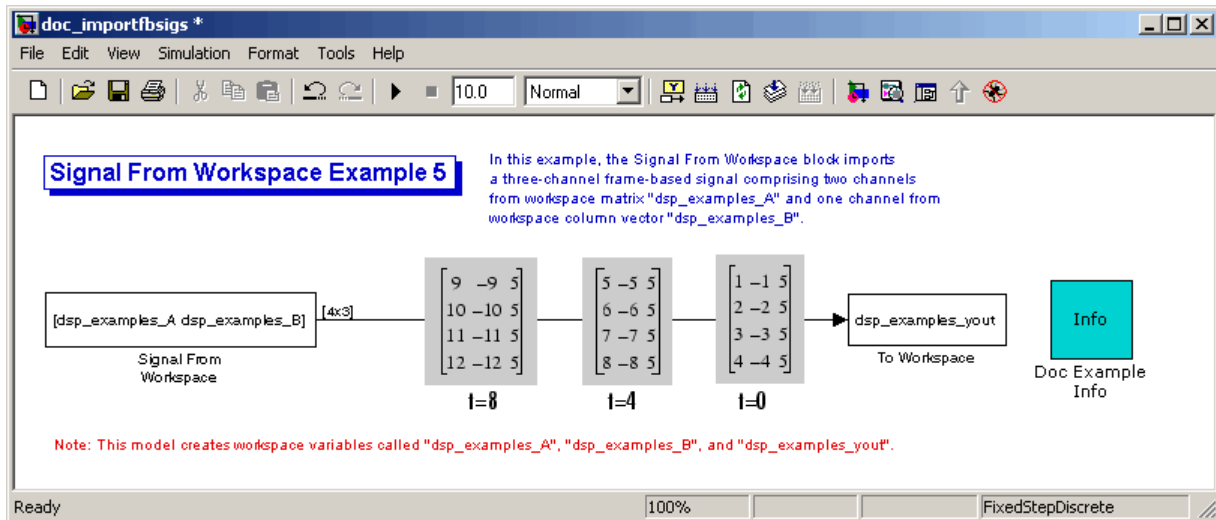
You have now successfully imported a three-channel frame-based signal into your model using the Signal From Workspace block.

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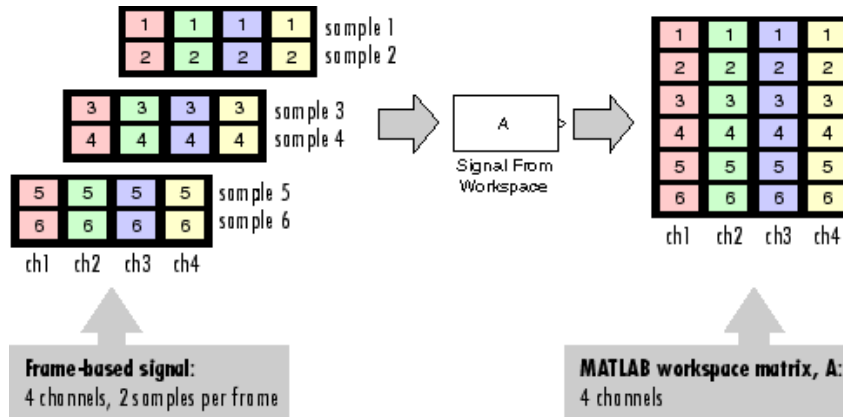
In the following example, you use a Signal To Workspace block to export a frame-based signal to the MATLAB workspace:

1 Open the Signal From Workspace Example 7 model by typing

```
doc_exportfbsigs
```

at the MATLAB command line.
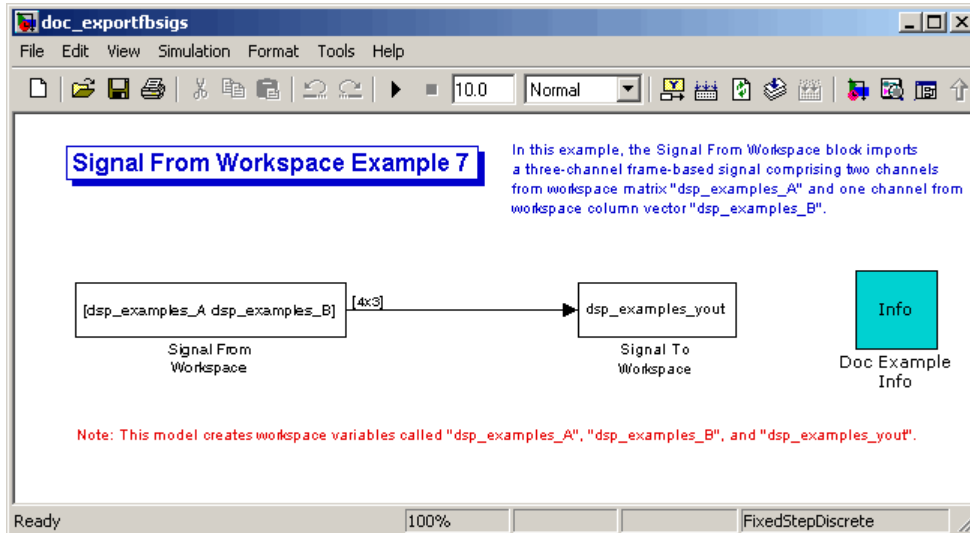
Also, the following variables are defined in the MATLAB workspace:

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 frame (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.
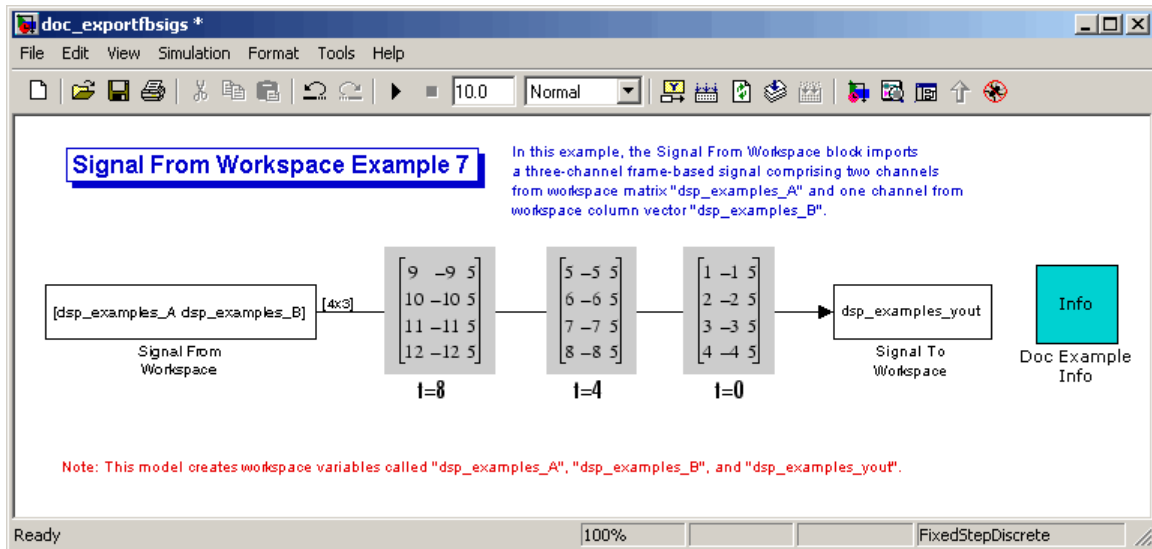
5 At the MATLAB command line, type dsp_examples_yout.

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.

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

When constructing a frame-based or multirate model, it is often helpful to check the rates that Simulink computes for different signals. The two basic ways to inspect the sample rates and frame rates in a Simulink model are the Probe block and sample time color coding. Use the Probe block if you want to view the sample or frame period of a signal. Use sample time color coding if you want to view the sample or frame rate of a signal.

This section includes the following topics:

- "Sample Rate and Frame Rate Concepts" on page 2-2 — Review the definitions of frame period, sample period, frame rate, and sample rate

- "Inspecting Sample-Based Signals Using the Probe Block" on page 2-4 — Display the sample period of a sample-based signal

- "Inspecting Frame-Based Signals Using the Probe Block" on page 2-6 — Display the frame period of a frame-based signal

- "Inspecting Sample-Based Signals Using Color Coding" on page 2-8 — Display the sample rate of a sample-based signal

- "Inspecting Frame-Based Signals Using Color Coding" on page 2-10 — Display the frame rate of a frame-based signal

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.
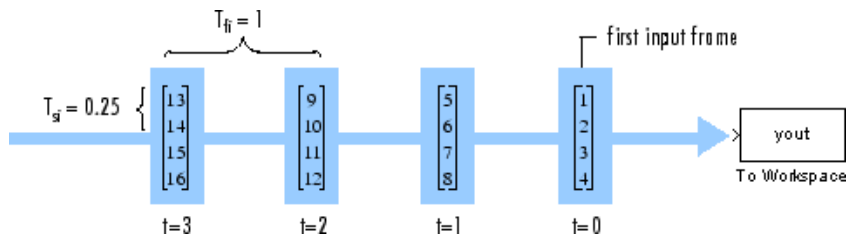
The frame rate of a signal is the reciprocal of the frame period. For instance, the input frame rate would be $1/T_{fi}$. Similarly, the output frame rate would be $1/T_{fo}$.

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 blocks from the Signal Processing Blockset 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.
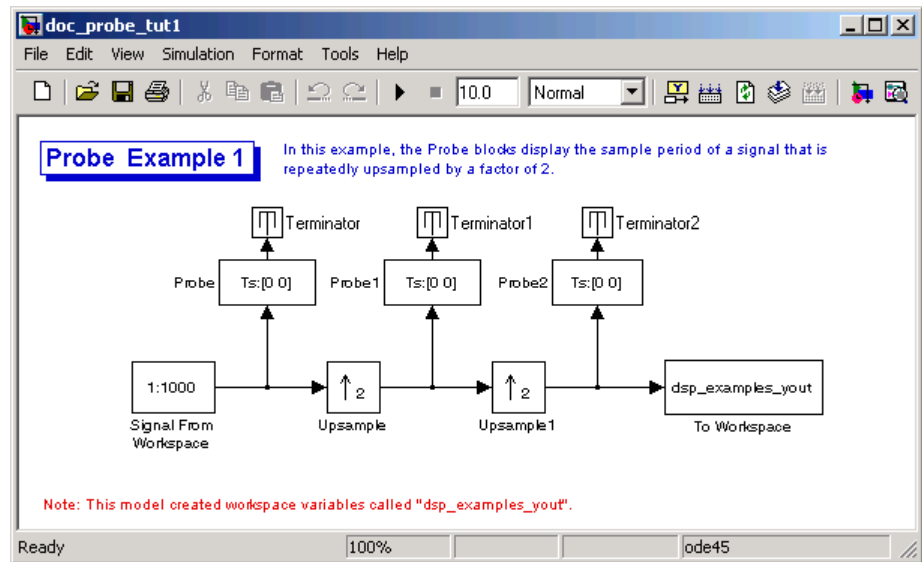
2 Run the model.

The figure below illustrates how the Probe blocks display the sample period of the signal before and after each upsample operation.
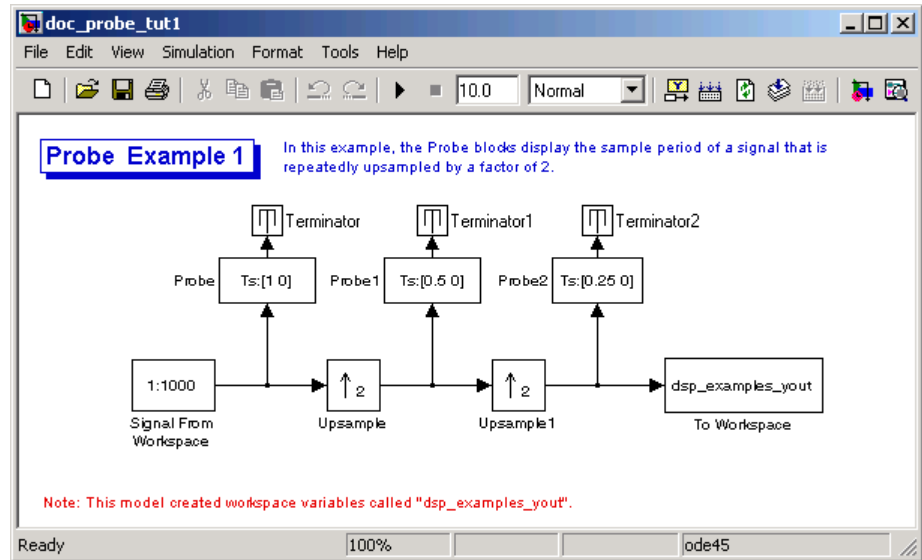
As displayed by the Probe blocks, the output from the Signal From Workspace block is a sample-based signal with a sample period of 1 second. The output from the first Upsample block has a sample period of 0.5 second, and the output from the second Upsample block has a sample period of 0.25 second.

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 blocks from the Signal Processing Blockset 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.
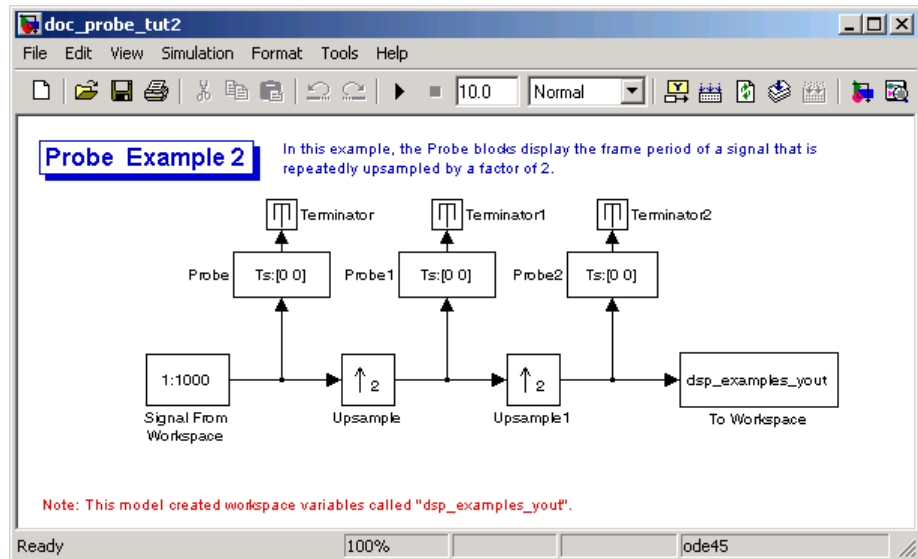
2 Run the model.

The figure below illustrates how the Probe blocks display the frame period of the signal before and after each upsample operation.
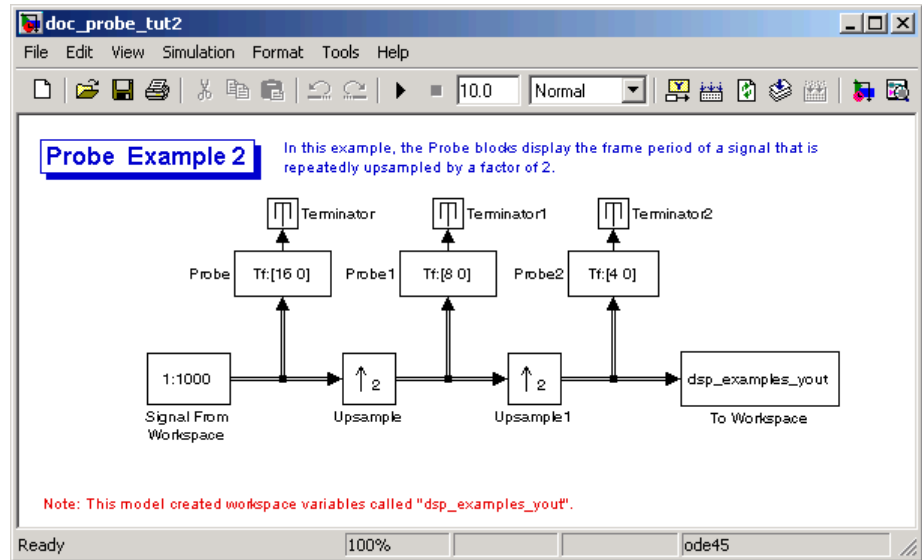
As displayed by the Probe blocks, the output from the Signal From Workspace block is a frame-based signal with a frame period of 16 seconds. The output from the first Upsample block has a frame period of 8 seconds, and the output from the second Upsample block has a sample period of 4 seconds.

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.
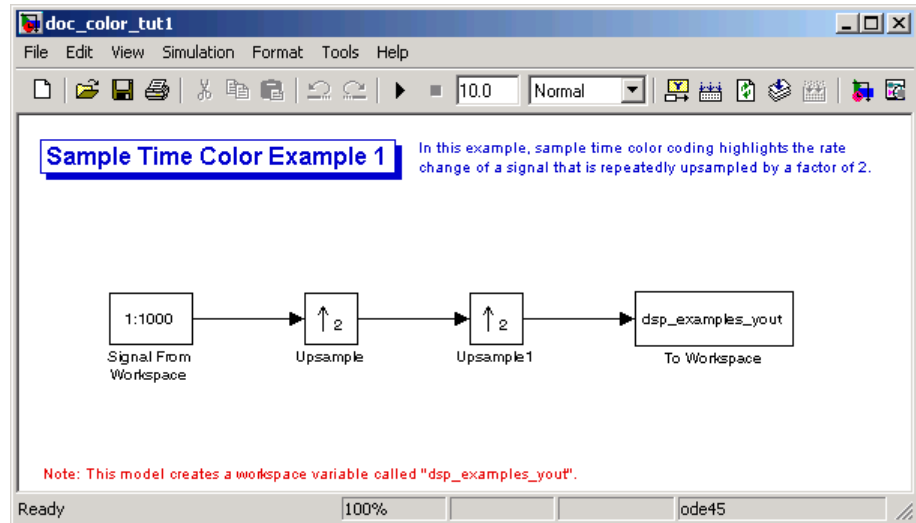
2 From the **Format** menu, point to **Port/Signal Displays**, and select **Sample Time Colors**.

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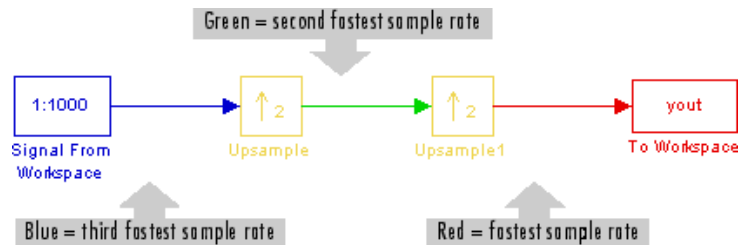
Every sample-based signal in this model has a different sample rate. Therefore, each signal is assigned a different color.

For more information about sample time color coding, see "Displaying Sample Time Colors" in the Using 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.
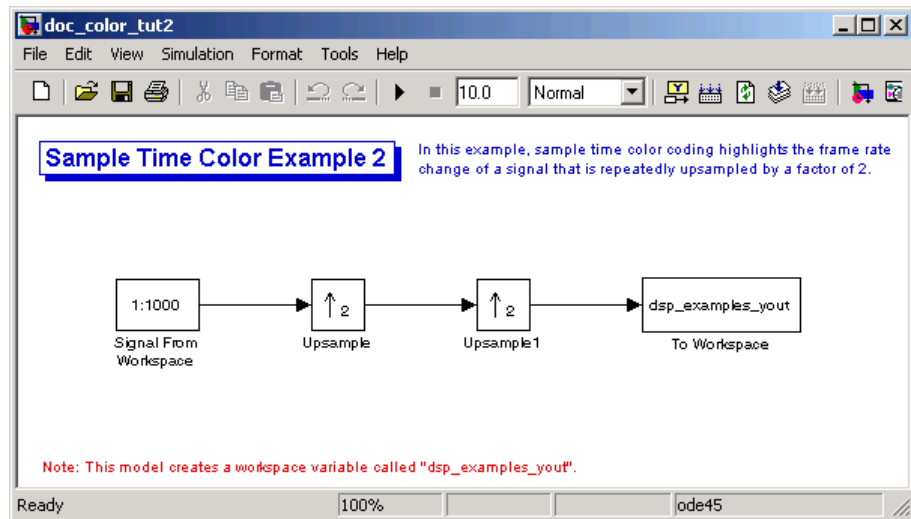
2 To turn on sample time color coding, from the **Format** menu, point to **Port/Signal Displays**, and select **Sample Time Colors**.

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.
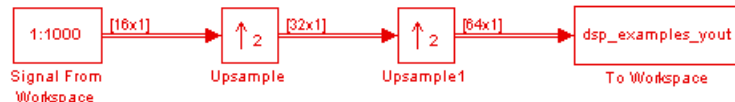
For more information about sample time color coding, see "Displaying Sample Time Colors" in the Using Simulink documentation.

Converting Sample and Frame Rates

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.

This section includes the following topics:

- "Rate Conversion Blocks" on page 2-12 — List of the principal rate conversion blocks in the Signal Processing Blockset

- "Rate Conversion by Frame-Rate Adjustment" on page 2-14 — Use the Downsample block to downsample a signal by changing its frame rate

- "Rate Conversion by Frame-Size Adjustment" on page 2-16 — Use the Downsample block to downsample a signal by changing its frame size

- "Avoiding Unintended Rate Conversion" on page 2-19 — Learn where rate conversions can occur in a model in order to avoid misleading results

- "Frame Rebuffering Blocks" on page 2-24 — List and descriptions of the principal frame rebuffering blocks in the Signal Processing Blockset

- "Buffering with Preservation of the Signal" on page 2-27 — Use the Buffer block to rebuffer a signal from a smaller to a larger frame size

- "Buffering with Alteration of the Signal" on page 2-29 — Use the Buffer block to rebuffer a signal from a smaller to a larger frame size using overlapping frames

Rate Conversion Blocks

The following table lists the principal rate conversion blocks in the Signal Processing Blockset. Blocks marked with an asterisk (*) offer the option of changing the rate by either adjusting the frame size or frame rate.

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

Direct Rate Conversion

Rate conversion blocks accept an input signal at one sample rate, and propagate the same signal at a new sample rate. Several of these blocks contain a **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 2-56. Also see "Models with Multiple Sample Rates" 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.
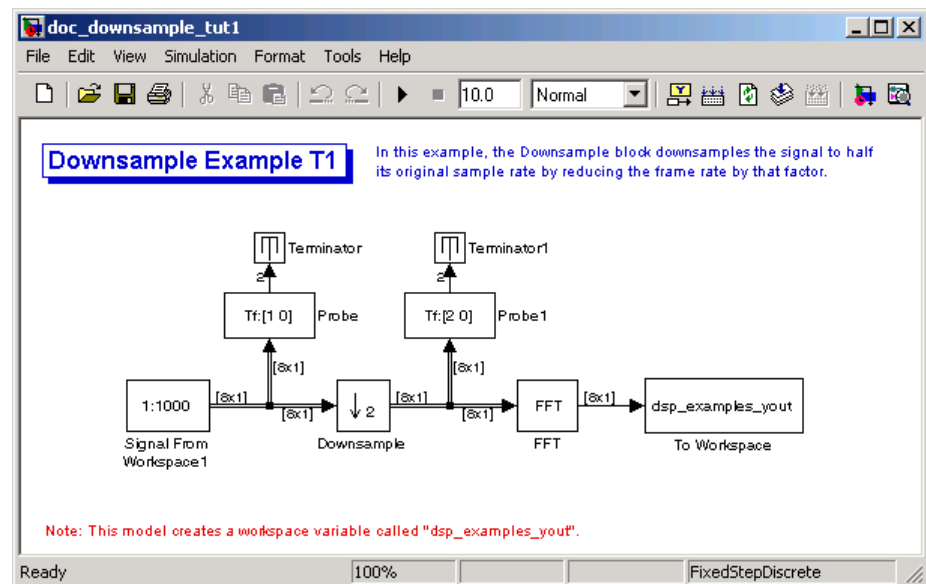
Because $T_{fi} = M_i \times T_{si}$, the input frame period, T_{fi}, is $T_{fi} = 8 \times 0.125 = 1$ second. This value is displayed by the first Probe block. Therefore the input frame rate, $1/T_{fi}$, is also 1 second.

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.
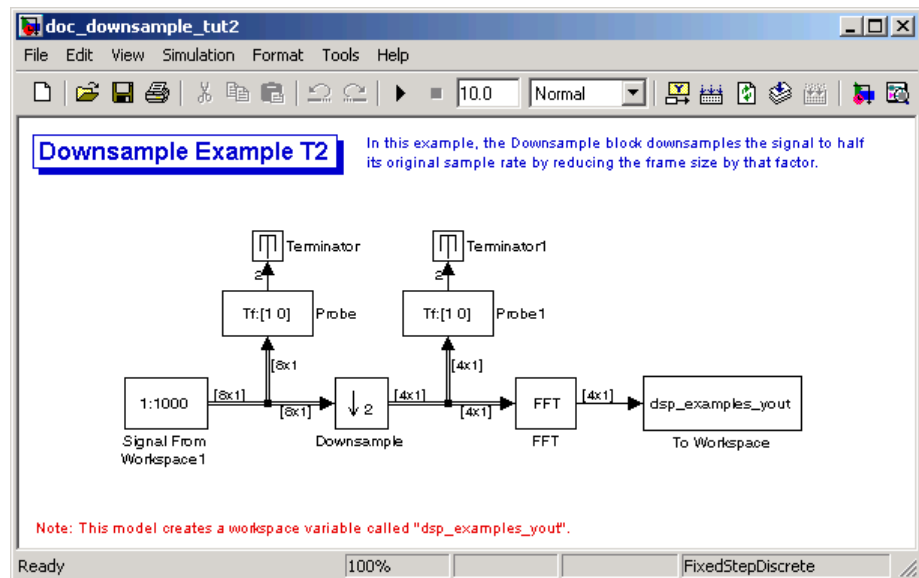
Because $T_{fi} = M_i \times T_{si}$, the input frame period, T_{fi}, is $T_{fi} = 8 \times 0.125 = 1$ second. This value is displayed by the first Probe block. Therefore the input

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**.

2-19

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.
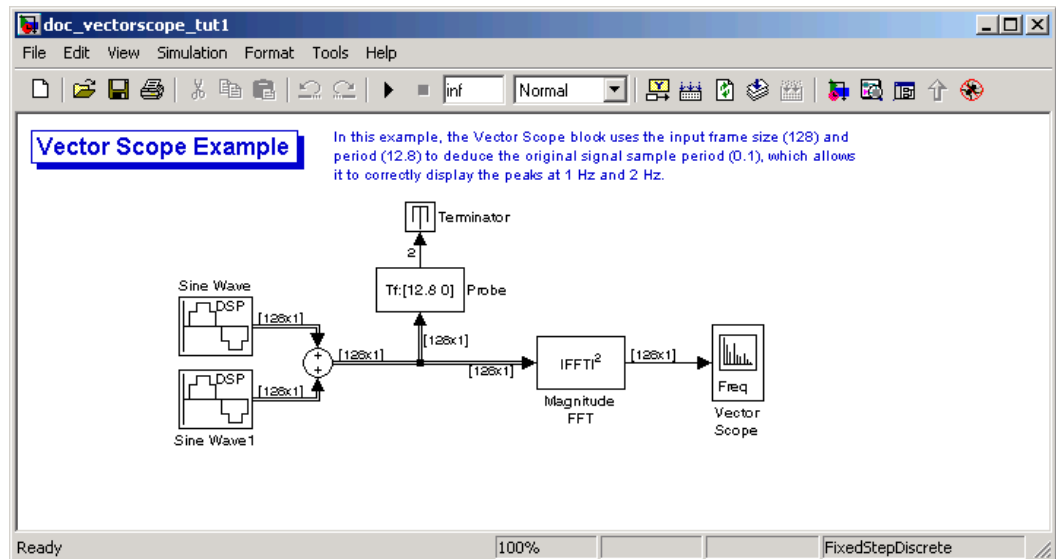
The **Vector Scope** window displays the magnitude FFT of a signal composed of two sine waves, with frequencies of 1 Hz and 2 Hz.
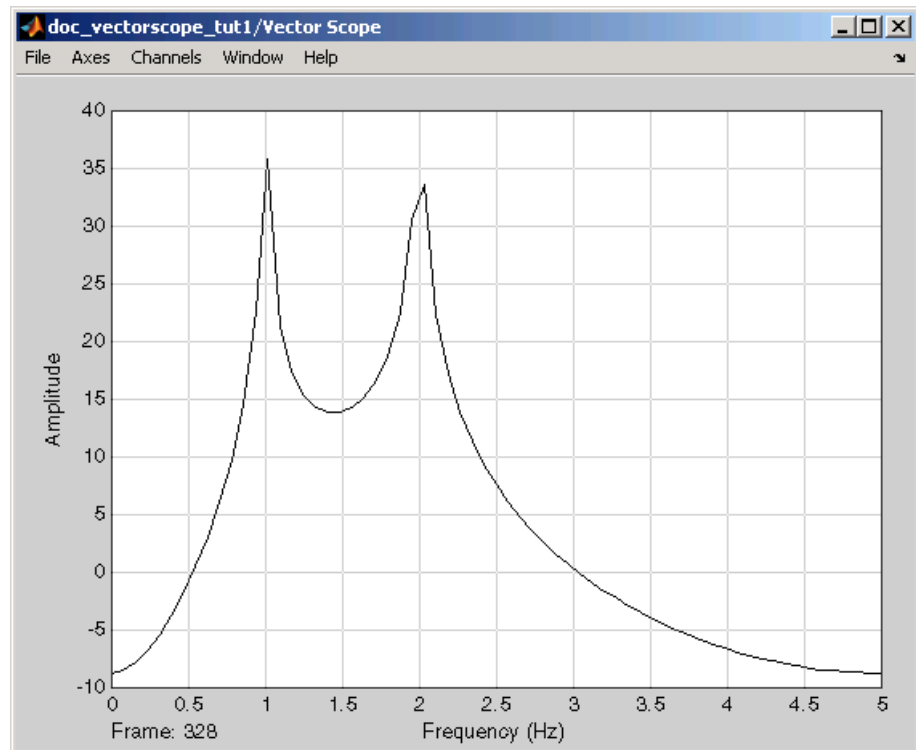
The Vector Scope block uses the input frame size (128) and period (12.8) to deduce the original signal's sample period (0.1), which allows it to correctly display the peaks at 1 Hz and 2 Hz.

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.
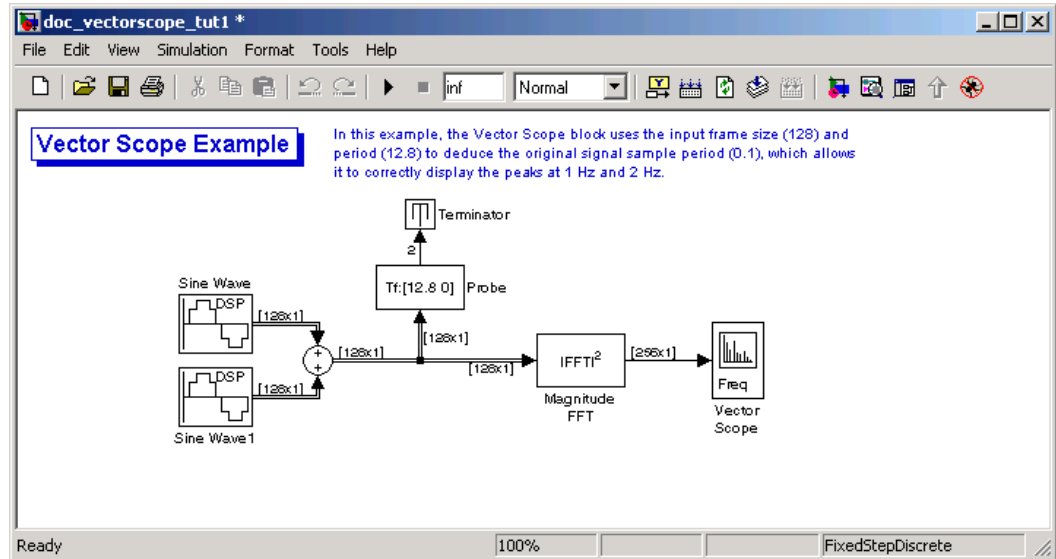
The **Vector Scope** window displays the magnitude FFT of a signal composed of two sine waves, with frequencies of 2 Hz and 4 Hz.
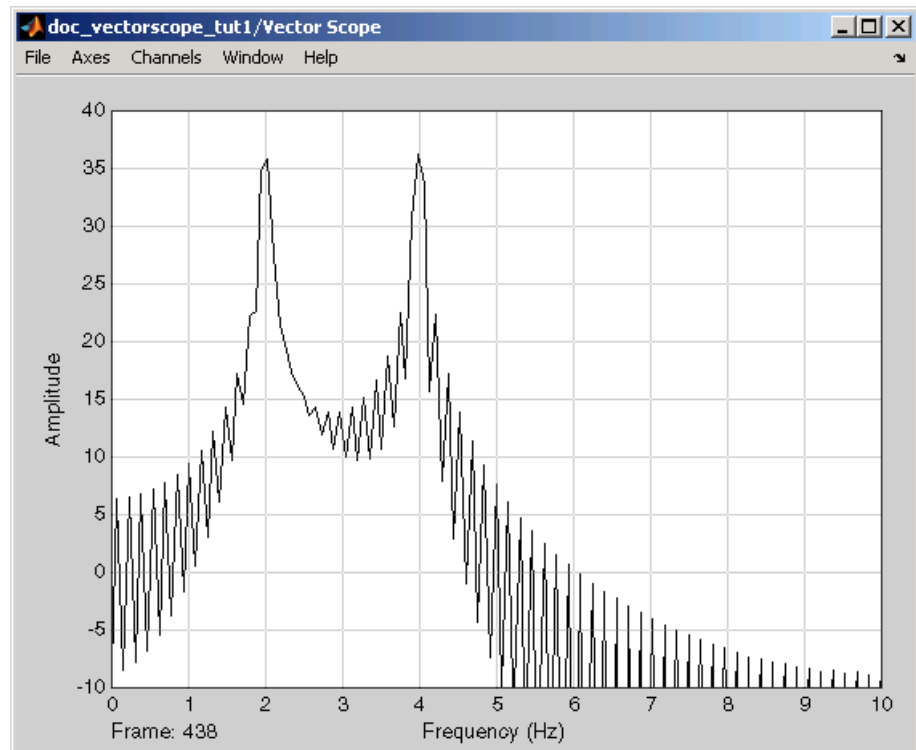
In this case, based on the input frame size (256) and frame period (12.8), the Vector Scope block incorrectly calculates the original signal's sample period to be (12.8/256) or 0.05 second. As a result, the spectral peaks appear incorrectly at 2 Hz and 4 Hz rather than 1 Hz and 2 Hz.

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

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 buffering blocks in the Signal Processing Blockset.

Block	Library
Buffer	Signal Management/ Buffers
Delay Line	Signal Management/ Buffers
Unbuffer	Signal Management/ Buffers
Variable Selector	Signal Management/ Indexing
Zero Pad	Signal Operations

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.

- The Zero Pad block adds samples to the sequence by appending zeros to each frame when **Pad along** is set to Columns. 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 **Number of output rows** parameter.

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 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. 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.
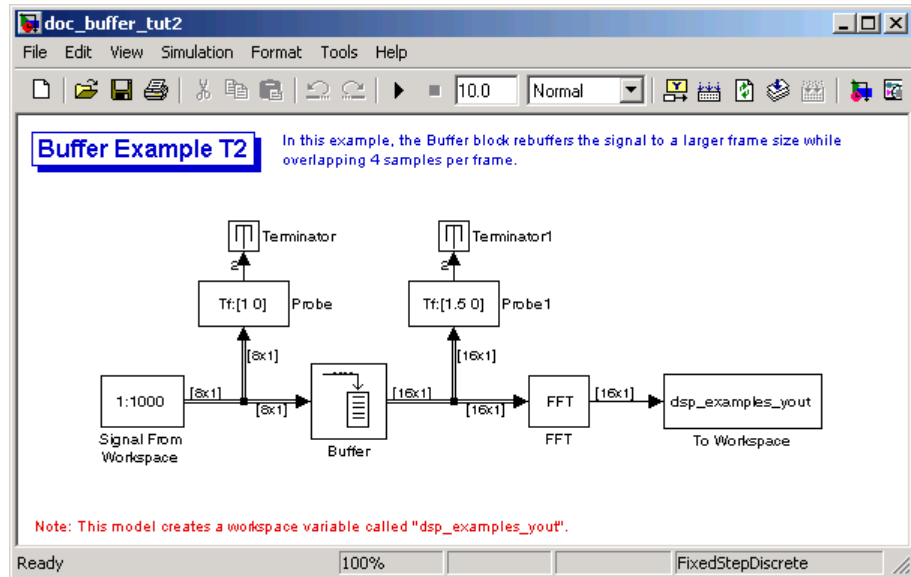
Note that the input to the Buffer block has a frame size of 8 and the output of the block has a frame size of 16. The relation for the output frame period for the Buffer block is

$$T_{fo} = (M_o - L)T_{si}$$

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

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.

This section includes the following topics:

- "Buffering Sample-Based Signals into Frame-Based Signals" on page 2-33 — Use the Buffer block to buffer a two-channel sample-based signal into a two-channel frame-based signal

- "Buffering Sample-Based Signals into Frame-Based Signals with Overlap" on page 2-36 — Use the Buffer block to buffer a four-channel, sample-based signal into a four-channel frame-based signal. Because of the buffer overlap, the input sample period is not conserved.

- "Buffering Frame-Based Signals into Other Frame-Based Signals" on page 2-40 — Use the Buffer block to buffer a two-channel frame-based signal with frame size 4 into a frame-based signal with frame size 3. Because of the buffer overlap, the input sample period is not conserved.

- "Buffering Delay and Initial Conditions" on page 2-42 — Learn how to use the rebuffer_delay function to calculate the delay introduced by the Buffer and Unbuffer blocks during multitasking operations

- "Unbuffering Frame-Based Signals into Sample-Based Signals" on page 2-43 — Use the Unbuffer block to unbuffer a two-channel frame-based signal into a two-channel sample-based signal

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.

sample 1, sample 2, sample 3, sample 4, sample 5, sample 6 — ch1 ch2 ch3 ch4

Multichannel sample-based signal:
4 channels

Multichannel frame-based signal:
4 channels, 6 samples per frame
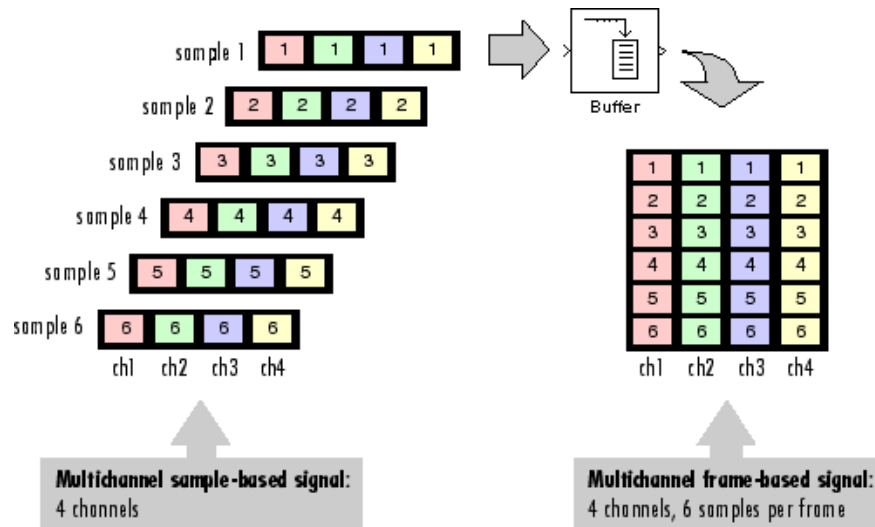
In the following example, a two-channel sample-based signal is buffered into a two-channel frame-based signal using a Buffer block:

1 At the MATLAB command prompt, type doc_buffer_tut.

The Buffer Example model opens.
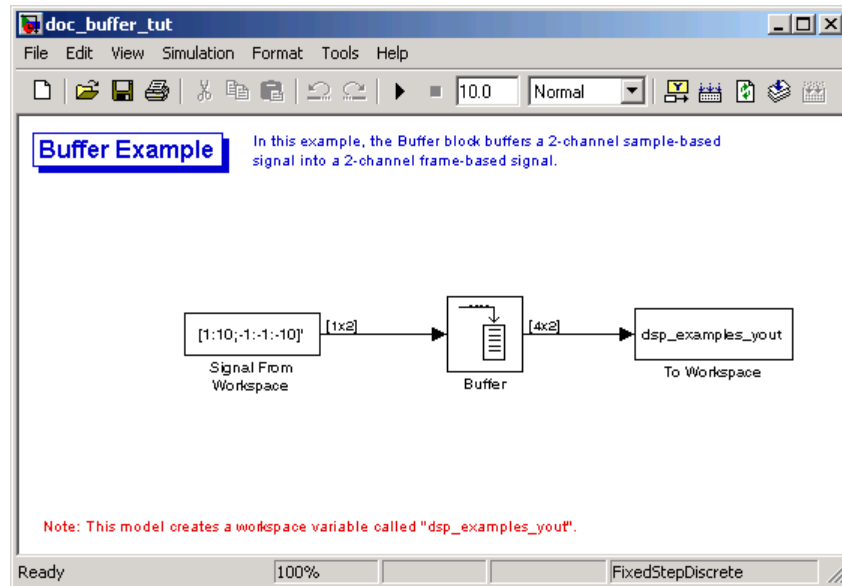
2 Double-click the Signal From Workspace block. The **Block Parameters: Signal From Workspace** dialog box opens.

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.
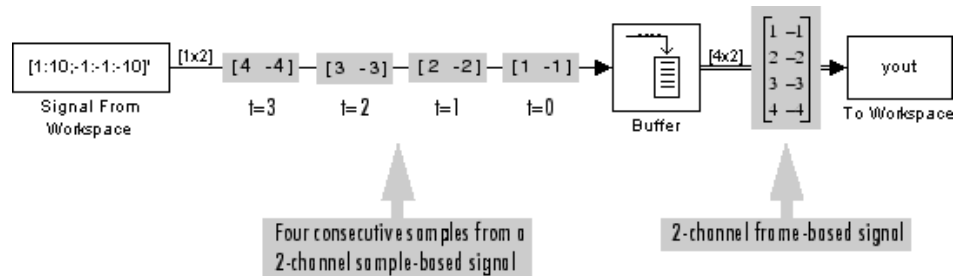
Note Alternatively, you can set the **Samples per frame** parameter of the Signal From Workspace block to 4 and create the same frame-based signal shown above without using a Buffer block. The Signal From Workspace block performs the buffering internally, in order to output a two-channel frame-based 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 $(M_o - L) * T_{si}$, where T_{si} is the input sample period and M_o is the **Buffer size**.

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

In the following example, a four-channel sample-based signal with sample period 1 is buffered to a frame-based signal with frame size 3 and frame period 2. Because of the buffer overlap, the input sample period is not conserved, and the output sample period is 2/3:

1 At the MATLAB command prompt, type `doc_buffer_tut3`.

The Buffer Example T3 model opens.
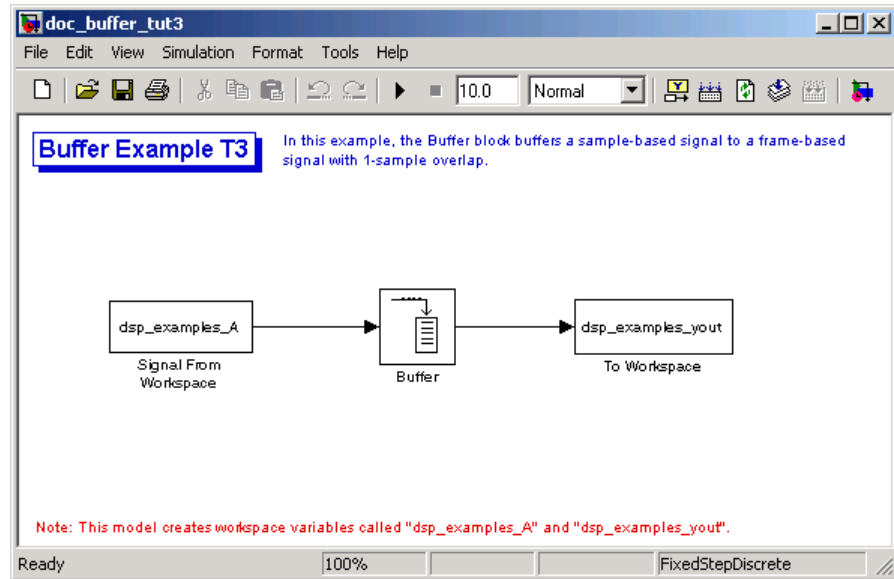
Also, the variable dsp_examples_A is loaded into the MATLAB workspace. This variable is defined as follows:

```
dsp_examples_A = [1 1 5 -1; 2 1 5 -2; 3 0 5 -3; 4 0 5 -4; 5 1 5 -5; 6 1
```

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** = dsp_examples_A
- **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.
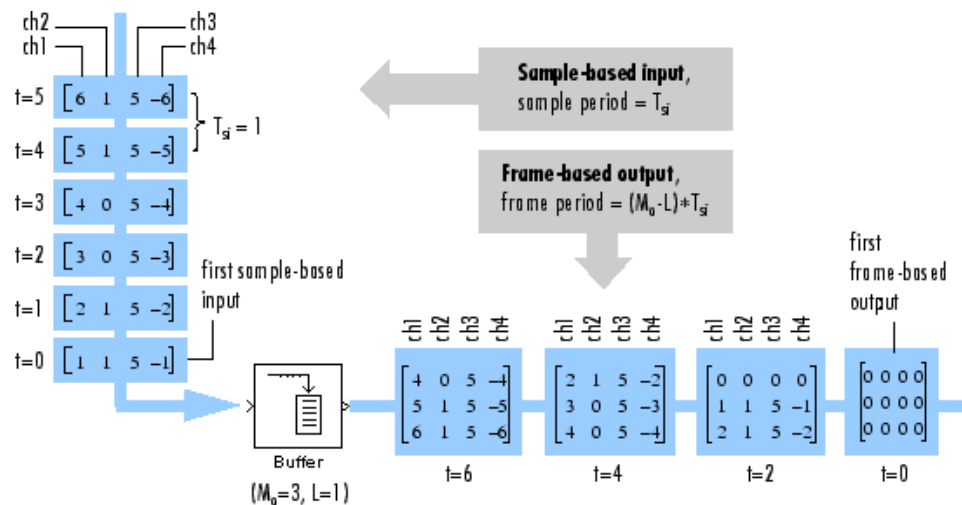
8 At the MATLAB command prompt, type dsp_examples_yout.

The following is displayed in the MATLAB Command Window.

```
dsp_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 2-56 for general information about algorithmic delay. For instructions on how to calculate buffering delay, see "Buffering Delay and Initial Conditions" on page 2-42.

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.
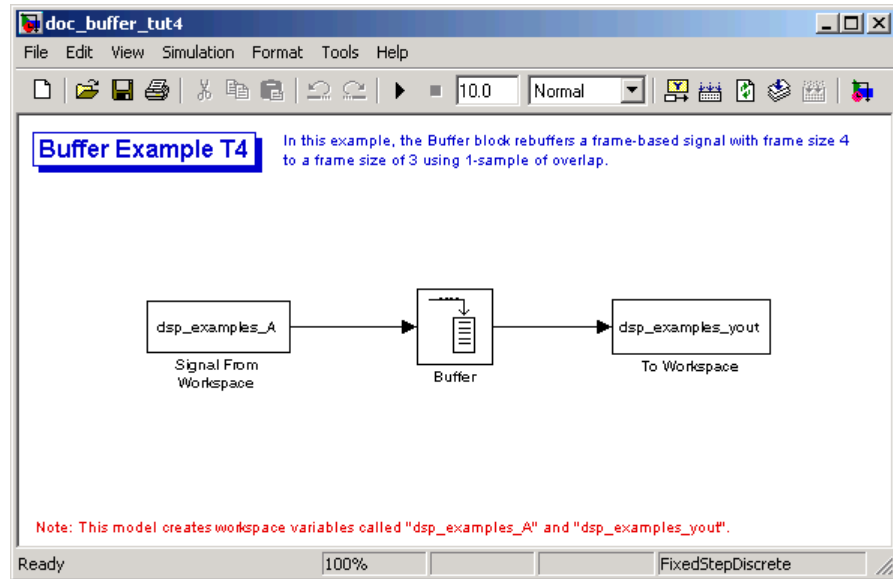
Also, the variable dsp_examples_A is loaded into the MATLAB workspace. This variable is defined as

```
dsp_examples_A = [1 1; 2 1; 3 0; 4 0; 5 1; 6 1; 7 0; 8 0]
```
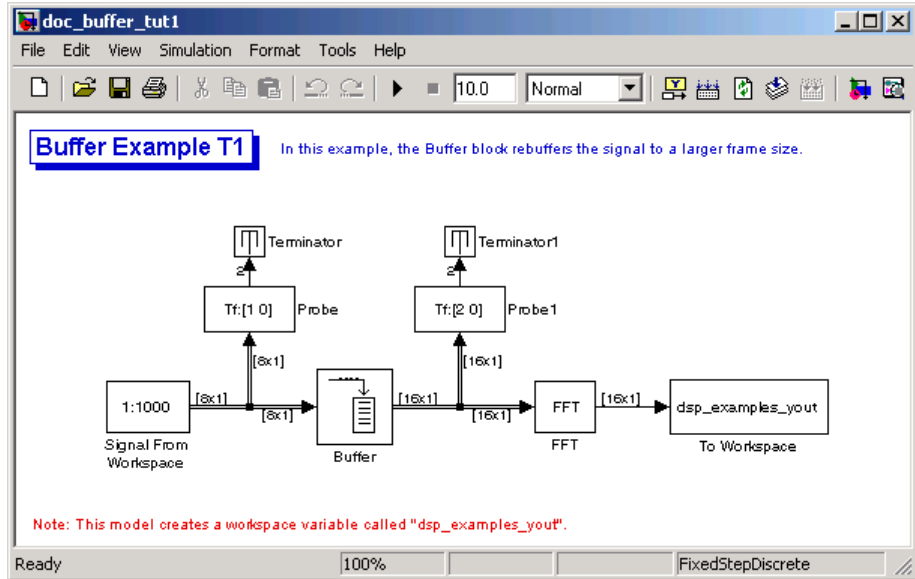
2 Double-click the Signal From Workspace block. The **Block Parameters: Signal From Workspace** dialog box opens.

3 Set the block parameters as follows:

- **Signal** = dsp_examples_A

- **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.
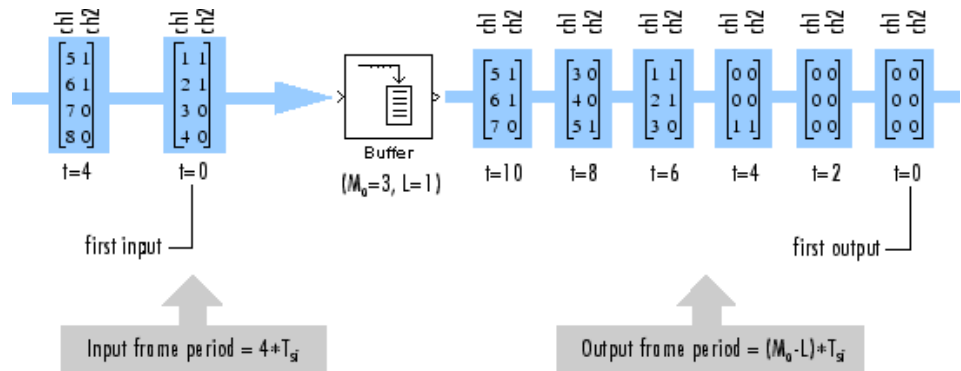
Note that the inputs do not begin appearing at the output until the last row of the third output matrix. This is due to the block's latency.

See "Excess Algorithmic Delay (Tasking Latency)" on page 2-56 for general information about algorithmic delay. For instructions on how to calculate buffering delay, and see "Buffering Delay and Initial Conditions" on page 2-42.

Buffering Delay and Initial Conditions

In the examples "Buffering Sample-Based Signals into Frame-Based Signals with Overlap" on page 2-36 and "Buffering Frame-Based Signals into Other

Frame-Based Signals" on page 2-40, 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,m)` 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 `m` 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 2-40 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 2-56. 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.
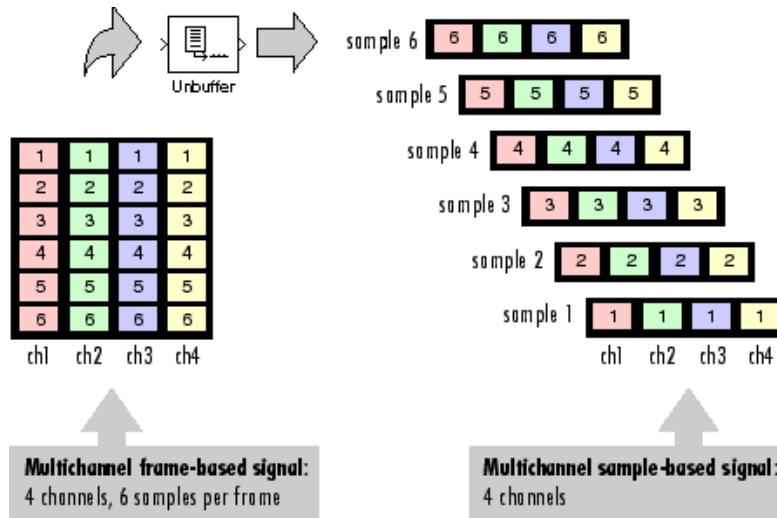
The sample period of the sample-based output, T_{so}, is related to the input frame period, T_{fi}, by the input frame size, M_i.

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

The Unbuffer block always preserves the signal's sample period ($T_{so} = T_{si}$). See "Converting Sample and Frame Rates" on page 2-12 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.
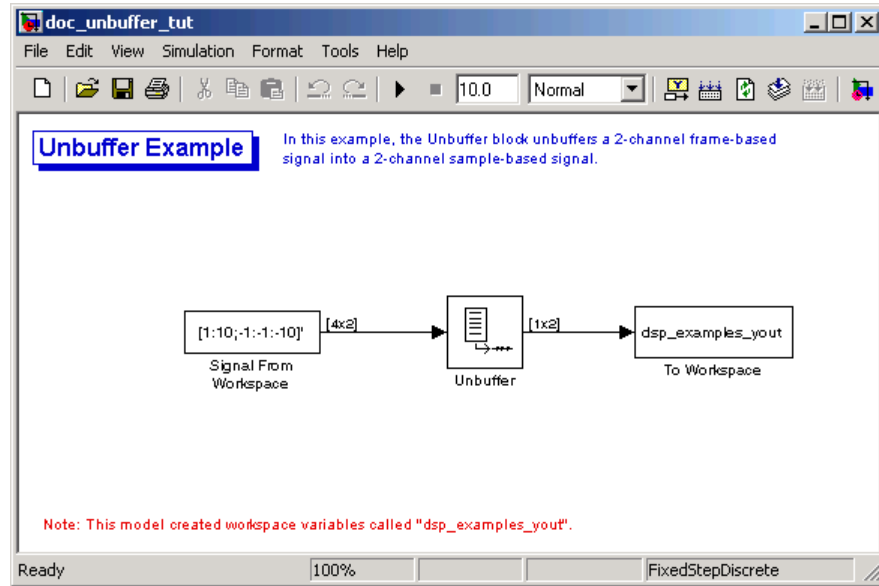
2 Double-click the Signal From Workspace block. The **Block Parameters: Signal From Workspace** dialog box opens.

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**.

2-45

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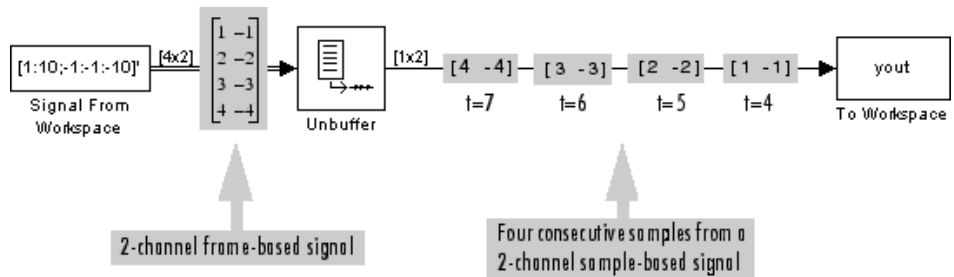
Note The Unbuffer block generates initial conditions not shown in the figure below with the value specified by the **Initial conditions** parameter. See the Unbuffer reference page for information about the number of initial conditions that appear in the output.

8 At the MATLAB command prompt, type dsp_examples_yout.

The following is a portion of the output.

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

    0    0


dsp_examples_yout(:,:,2) =

    0    0


dsp_examples_yout(:,:,3) =
```

```
     0      0


dsp_examples_yout(:,:,4) =

     0      0


dsp_examples_yout(:,:,5) =

     1     -1


dsp_examples_yout(:,:,6) =

     2     -2


dsp_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

The two types of delay that affect Simulink models are computational delay and algorithmic delay. This section explains the cause of each variety of delay. It describes how you can configure Simulink to minimize delay and increase simulation performance. It also discusses how to accurately predict the tasking latency of a particular model.

This section includes the following topics:

- "Computational Delay" on page 2-48 — Learn the cause of computational delay and how to reduce it

- "Algorithmic Delay" on page 2-50 — Learn the cause of algorithmic delay

- "Zero Algorithmic Delay" on page 2-50 — Work with a block that has no algorithmic delay

- "Basic Algorithmic Delay" on page 2-53 — Work with a block that has algorithmic delay

- "Excess Algorithmic Delay (Tasking Latency)" on page 2-56 — Explore the block and model characteristics that can affect the tasking latency of a particular block

- "Predicting Tasking Latency" on page 2-58 — Use the Upsample block's block reference page to predict the tasking latency of a model

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 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 MATLAB 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 Using 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 1-17 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 Getting Started Signal Processing Blockset documentation.

- Turn off the Simulink status bar by deselecting the **Status bar** option in the **View** menu. Simulation speed will improve, but the time indicator will not be visible.

- Run your simulation from the MATLAB command line by typing

    ```
    sim(gcs)
    ```

    This method of starting a simulation can greatly increase the simulation speed, but also has several limitations:

    - You cannot interact with the simulation (to tune parameters, for instance).

- You must press **Ctrl+C** to stop the simulation, or specify start and stop times.

- There are no graphics updates in M-file S-functions, which include blocks such as Vector Scope, etc.

- Use Real-Time Workshop® 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 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 2-50

- "Basic Algorithmic Delay" on page 2-53

- "Excess Algorithmic Delay (Tasking Latency)" on page 2-56

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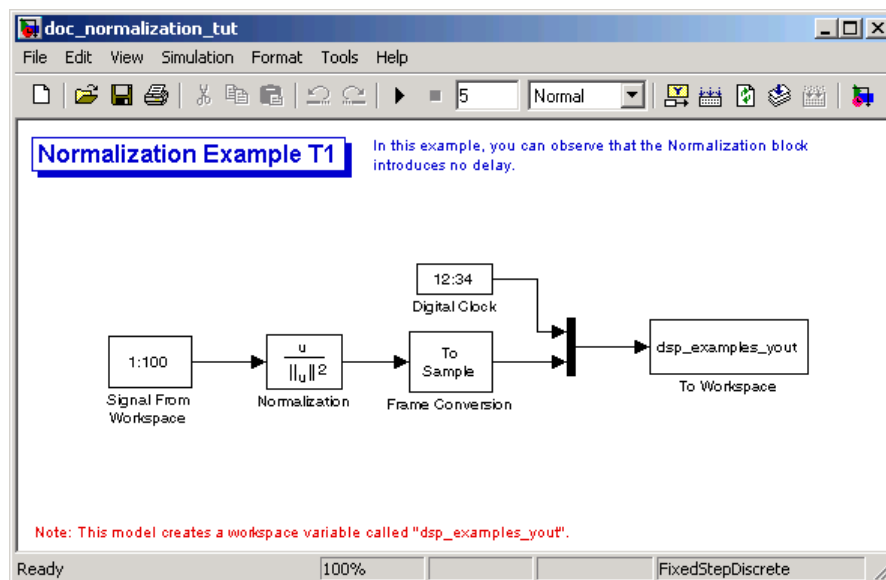
2 Double-click the Signal From Workspace block. The **Block Parameters: Signal From Workspace** dialog box opens.

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. See the Using Simulink documentation for more information about "Algebraic Loops".

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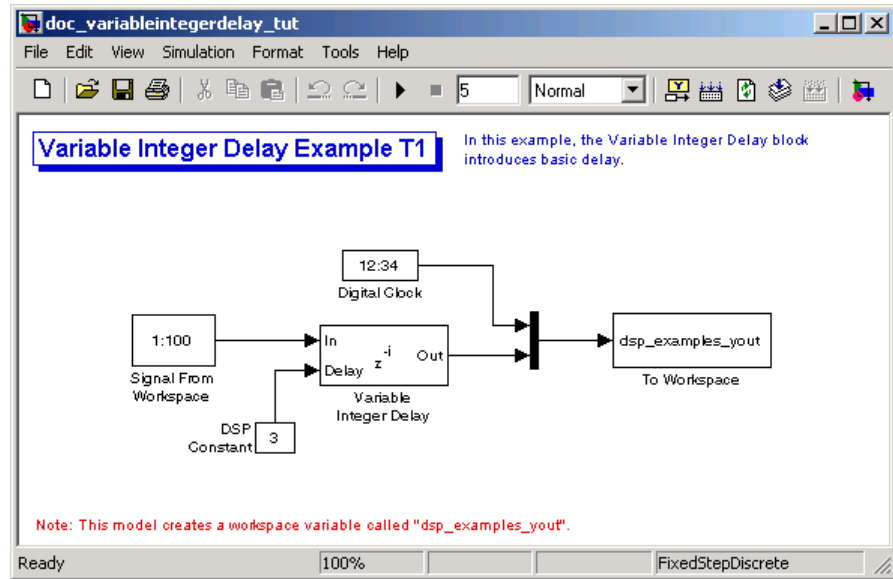
2 Double-click the Signal From Workspace block. The **Block Parameters: Signal From Workspace** dialog box opens.

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 DSP Constant block. The **Block Parameters: DSP Constant** dialog box opens.

6 Set the **Constant value** parameter to 3, and then click **OK**.

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.
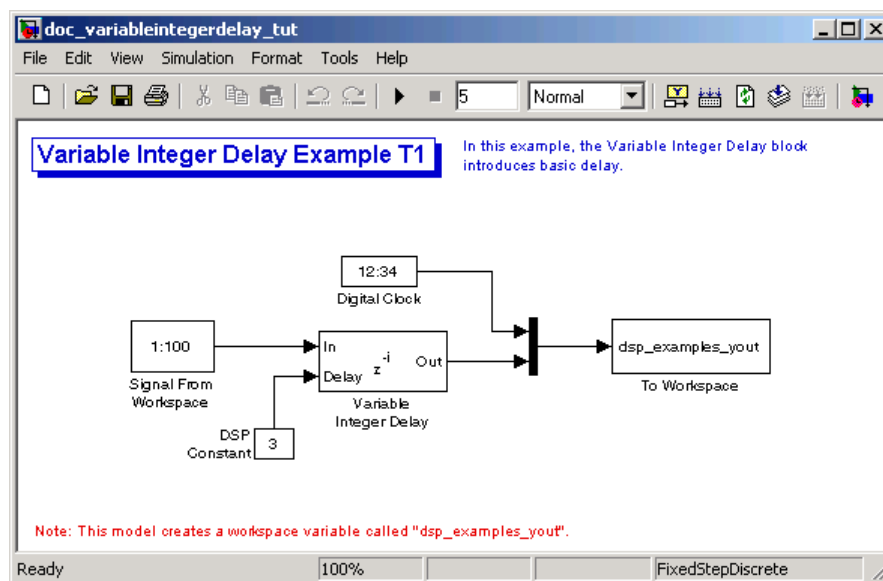
11 At the MATLAB command prompt, type dsp_examples_yout

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 blocks in the Signal Processing Blockset 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

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 "Models with Multiple Sample Rates" in the Real-Time Workshop User's Guide documentation.

Block Rate Type

A block is called single-rate when all of its input and output ports operate at the same frame rate. A block is called multirate when at least one input or output port has a different frame rate than the others.

Many blocks are permanently single-rate. This means that all input and output ports always have the same frame rate. For other blocks, the block parameter settings determine whether the block is single-rate or multirate. Only multirate blocks are subject to tasking latency.

Note Simulink may report an algebraic loop error if it detects a feedback loop composed entirely of multirate blocks. To break such an algebraic loop, insert a single-rate block with nonzero delay, such as a Unit Delay block. See the Using 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.
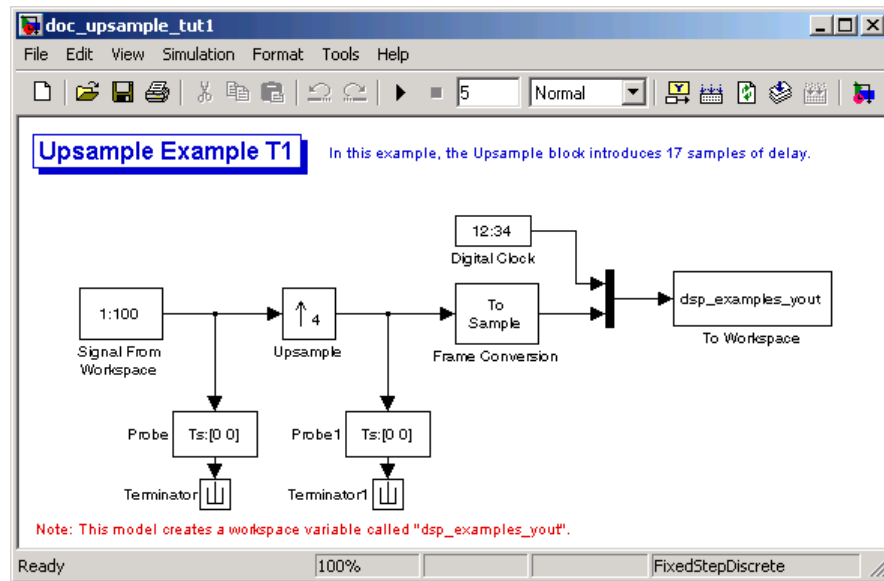
2 From the **Simulation** menu, select **Configuration Parameters**.

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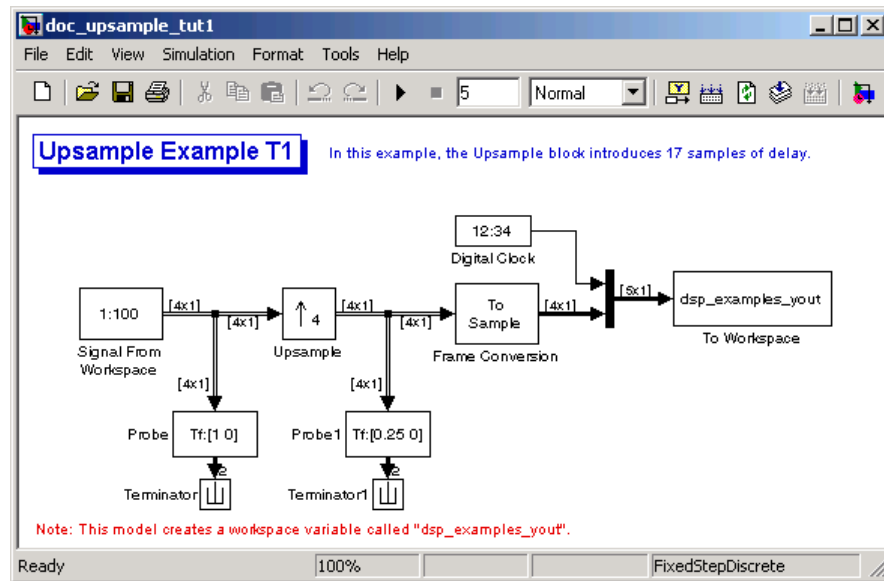
The model prepends the current value of the Simulink timer, from the Digital Clock block, to each output frame. The Frame Conversion block converts the frame-based signal into a sample-based signal so that the output in the MATLAB Command Window is easily readable.

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 =
```

0	-1.0000	0	0	0	1st output frame
0.2500	-1.0000	0	0	0
0.5000	-1.0000	0	0	0
0.7500	-1.0000	0	0	0
1.0000	1.0000	0	0	0	5th output frame
1.2500	2.0000	0	0	0
1.5000	3.0000	0	0	0
1.7500	4.0000	0	0	0
2.0000	5.0000	0	0	0

time

"Latency and Initial Conditions" in the Upsample block's reference page indicates that when Simulink is in multitasking mode, the first sample of the block's frame-based input appears in the output as sample M_iL+D+1, where M_i is the input frame size, L is the **Upsample factor**, and D is the **Sample offset**. This formula predicts that the first input in this example should appear as output sample 17 (that is, 4*4+0+1).

The first column of the output is the Simulink time provided by the Digital Clock block. The four values to the right of each time are the values in the output frame at that time. You can see that the first sample in each of the first four output frames inherits the value of the Upsample block's **Initial conditions** parameter. As a result of the tasking latency, the first input value appears as the first sample of the 5th output frame (at *t*=1). This is sample 17.

Now try running the model in single-tasking mode.

15 From the **Simulation** menu, select **Configuration Parameters**.

16 In the **Solver** pane, from the **Type** list, select `Fixed-step`. From the **Solver** list, select `discrete (no continuous states)`.

17 From the **Tasking mode for periodic sample times** list, select `SingleTasking`.

18 Run the model.

The model now runs in single-tasking mode.

19 At the MATLAB command prompt, type squeeze(dsp_examples_yout)'.

The first few samples of the result, ans, are:

```
ans =
```

time					
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	

"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.

3

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

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.

Required Parameters

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.

This section includes the following topics:

- "Implementing a Lowpass Filter" on page 3-3 — Create a lowpass filter using the Digital Filter block

- "Implementing a Highpass Filter" on page 3-4 — Create a highpass filter using the Digital Filter block

- "Filtering High-Frequency Noise" on page 3-5 — Build a system capable of filtering high-frequency noise using a highpass and lowpass filter

- "Specifying Static Filters" on page 3-10 — Use the Digital Filter block to create a static filter

- "Specifying Time-Varying Filters" on page 3-11 — Use the Digital Filter block to create a time-varying filter

- "Specifying the SOS Matrix (Biquadratic Filter Coefficients)" on page 3-16 — Use the Digital Filter block to create a static biquadratic direct form II transposed filter

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 Designs 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**:

- **Transfer function type** = FIR (all zeros)

- **Filter structure** = Direct form transposed

- **Coefficient source** = Specify via dialog

- **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 the Signal Processing Toolbox or the Filter Design Toolbox, 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 3-18.

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_probe_tut1
```

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 Designs 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**:

- **Transfer function type** = FIR (all zeros)
- **Filter structure** = Direct form transposed
- **Coefficient source** = Specify via dialog
- **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 the Signal Processing Toolbox or the Filter Design Toolbox, 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 3-18.

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
Matrix Concatenation	Math Functions / Matrices and Linear Algebra / Matrix Operations	1
Random Source	Signal Processing Sources	1
Sine Wave	Signal Processing Sources	1
Sum	Simulink / Math Operations library	1
Vector 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
Matrix Concatenation	• **Number of inputs** = 3 • **Concatenation method** = Horizontal
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

Block	Parameter Setting
Sum	• **Icon shape** = Time • **List of signs** = ++
Vector Scope	**Scope Properties:** • **Input domain** = Time • **Time display span (number of frames)** = 1

5 Connect the blocks and label your signals as shown in the following figure. You need to resize some of your blocks to accomplish this task.
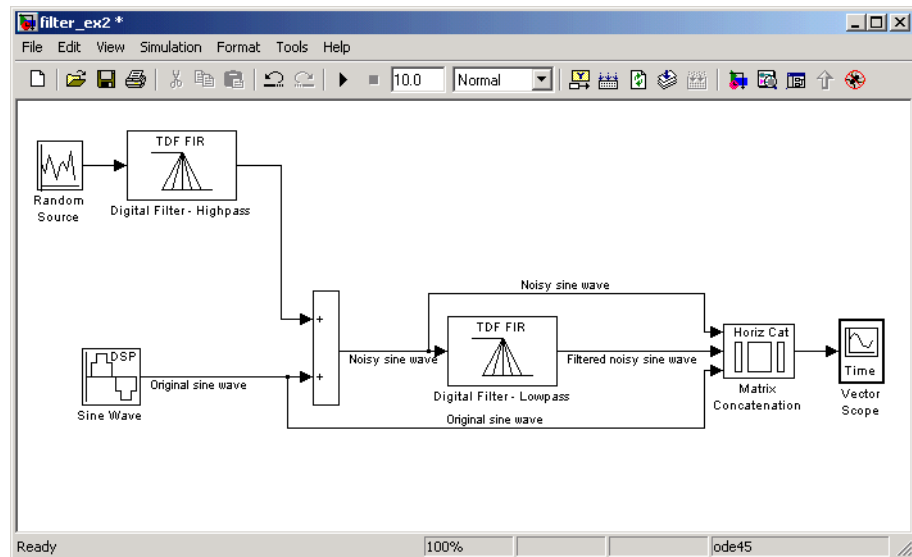
6 From the Simulation menu, select **Configuration Parameters**.

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 Double-click the Vector Scope block and click the **Display Properties** tab. Select the **Channel legend** check box and click **OK**. Next time you run the simulation, a legend appears in the Vector Scope window.

You can also set the color, style, and marker of each channel.

10 In the Vector Scope window, from the **Channels** menu, point to **Ch 1** and set the **Style** to **-**, **Marker** to **None**, and **Color** to **Black**.

Point to **Ch 2** and set the **Style** to **-**, **Marker** to **Diamond**, and **Color** to **Red**.

Point to **Ch 3** and set the **Style** to **None**, **Marker** to *****, and **Color** to **Blue**.
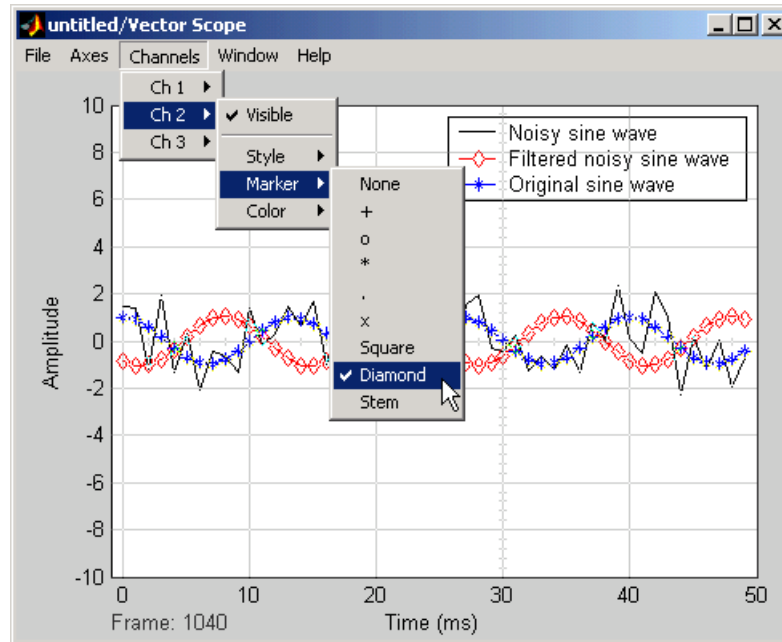
11 Rerun the simulation and compare the original sine wave, noisy sine wave, and filtered noisy sine wave in the **Vector Scope** display.

You can see that the lowpass filter filters out the high-frequency noise in the noisy sine wave.
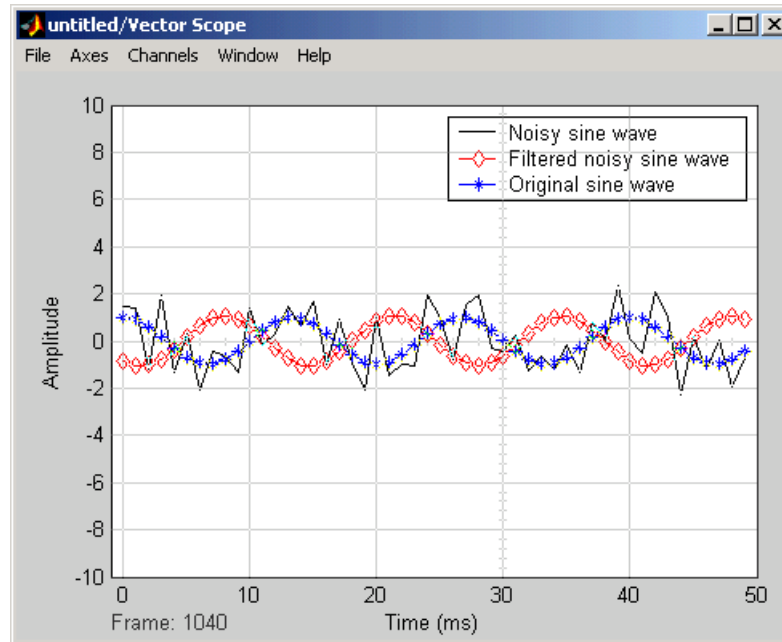
You have now used Digital Filter blocks to build a model that removes high frequency noise from a signal. For more information about designing and implementing a new filter, see "Digital Filter Design Block" on page 3-18.

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" on page 10-238:

- **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. To learn about SOS matrices, see "Specifying the SOS Matrix (Biquadratic Filter Coefficients)" on page 3-16.

- **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. The following diagram shows one block with an extra port for reflection coefficients, and another with extra ports for numerator and denominator coefficients.
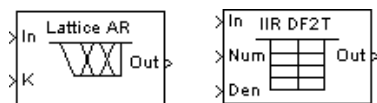
2 Set the **Coefficient update rate** parameter to `One filter per frame` or `One filter per sample` depending on how often you want to update the filter coefficients. To learn more, see "Setting the Coefficient Update Rate" on page 3-12.

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 3-13.

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 3-15.

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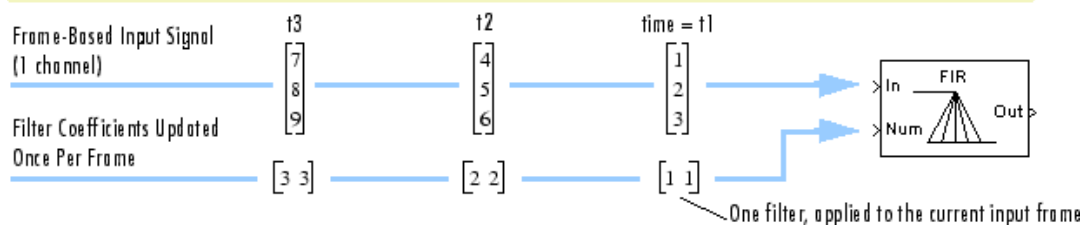
Update filter coefficients once per sample:
At time t1, the block applies filter [1 1] to the first sample in the first frame of data, filter [2 2] to the second sample, and filter [3 3] to the third sample. At time t2, the block updates the filter for each sample in the next input frame, and applies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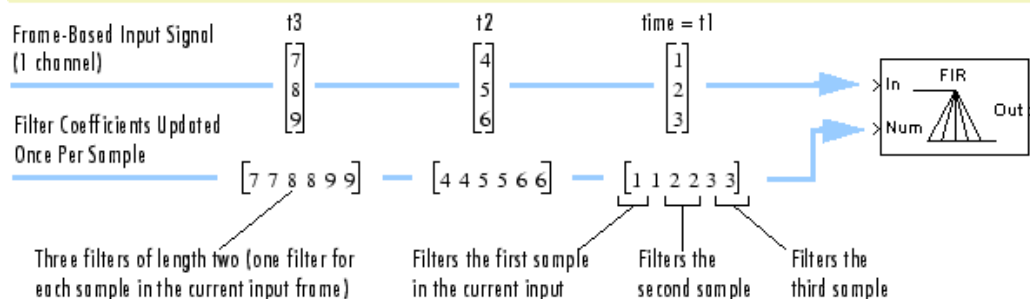
Providing Filter Coefficient Vectors at Block Input Ports

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 3-14.

- 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: [b0, b1, b2, ..., bn], [a0, a1, a2, ..., am], or [k1, k2, ..., kn]	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 $$\left[b_0 \ b_1 \ B_0 \ B_1 \ \beta_0 \ \beta_1 \right]$$ where $\left[b_0 \ b_1 \right]$ filters the first sample in the input frame, $\left[B_0 \ B_1 \right]$ 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.

Rate Requirements for Time-Varying Filter Coefficient Vectors (Continued)

Input Signal	Time-Varying Filter Coefficient Vectors	Rate Requirements (Also see the previous figure)
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.

The block omits this term in the structure when you set the
First denominator coefficient = 1, remove a0 term in the structure parameter.
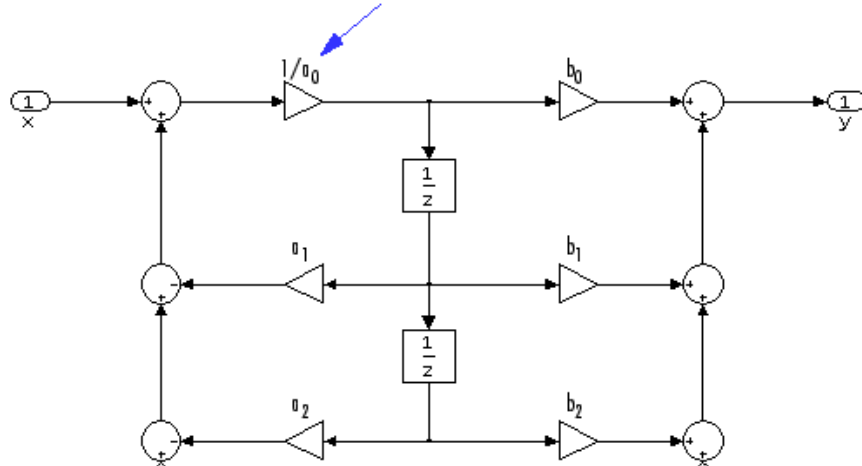
Specifying the SOS Matrix (Biquadratic Filter Coefficients)

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), 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.
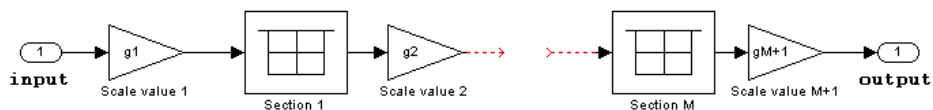
You can use the `ss2sos` and `tf2sos` functions from the Signal Processing Toolbox to convert a state-space or transfer-function description of your filter into the second-order section description used by this block.

$$
\begin{bmatrix}
b_{11} & b_{21} & b_{31} & a_{11} & a_{21} & a_{31} \\
b_{12} & b_{22} & b_{32} & a_{12} & a_{22} & a_{32} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
b_{1M} & b_{2M} & b_{3M} & a_{1M} & a_{2M} & a_{3M}
\end{bmatrix}
$$

Note The block normalizes each row by a_{1i} to ensure a value of 1 for the zero-delay denominator coefficients.

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 Getting Started Signal Processing Blockset documentation.

This section includes the following topics:

- "Overview of the Digital Filter Design Block" on page 3-19 — Learn the basic functionality of the Digital Filter Design block

- "Choosing Between Filter Design Blocks" on page 3-20 — Determine whether the Digital Filter Design block or the Filter Realization Wizard is right for your application

- "Creating a Lowpass Filter" on page 3-22 — Use the Digital Filter Design block to design and implement a lowpass filter

- "Creating a Highpass Filter" on page 3-24 — Use the Digital Filter Design block to design and implement a highpass filter

- "Filtering High-Frequency Noise" on page 3-26 — Create a system capable of filtering high-frequency noise using a highpass and a lowpass filter

Alternatively, you can use other MathWorks products, such as the Signal Processing Toolbox and Filter Design Toolbox, to design your filters. Once you design a filter using either toolbox, you can use one of the Signal Processing Blockset's filter implementation blocks, 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.

Overview of the Digital Filter Design Block

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` function in the Filter Design Toolbox and the `filter` function in MATLAB.

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 3-20.

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 Getting Started Signal Processing Blockset documentation.
  - 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 Sum, Gain, and Delay blocks from Simulink, 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/ISO C code that implements floating-point filters for embedded systems. For more information on code generation, see "Code Generation" in the Getting Started Signal Processing Blockset documentation.
- 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 3-32 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 Designs 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:
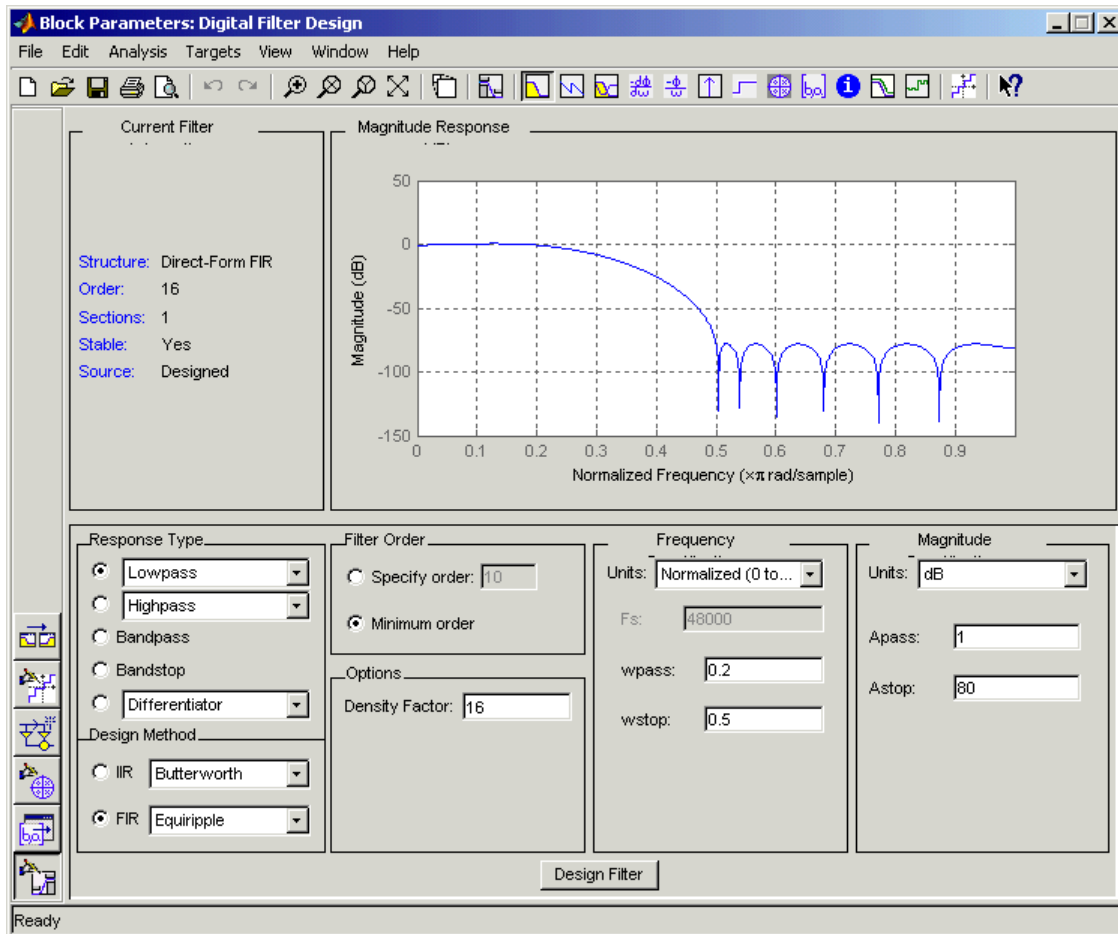
5 Click **Design Filter** at the bottom of the GUI to design the filter.

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 3-24, 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 3-22 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 Designs 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.
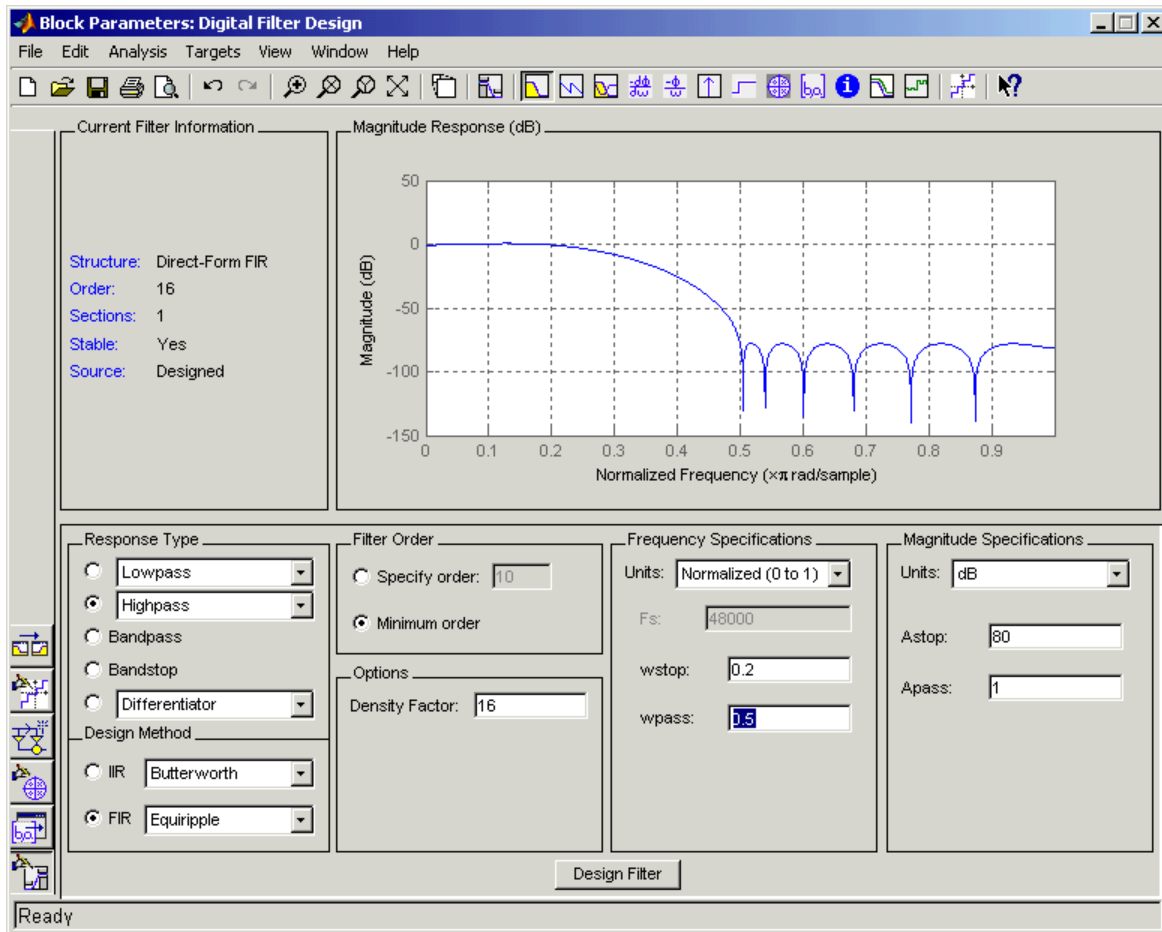
5 Click the **Design Filter** button at the bottom of the GUI to design the filter.

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 3-22. 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 3-26, 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 3-24 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 file.

Block	Library	Quantity
Matrix Concatenation	Math Functions / Matrices and Linear Algebra / Matrix Operations	1
Random Source	Signal Processing Sources	1

Block	Library	Quantity
Sine Wave	Signal Processing Sources	1
Sum	Simulink Math Operations library	1
Vector 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
Matrix Concatenation	• **Number of inputs** = 3 • **Concatenation method** = Horizontal
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

Parameter Settings for the Other Blocks (Continued)

Block	Parameter Setting
Sum	• **Icon shape** = rectangular
	• **List of signs** = ++
	• **Input domain** = Time
	• **Time display span (number of frames)** = 1
Vector Scope	**Scope Properties:**
	• **Input domain** = Time
	• **Time display span (number of frames)** = 1

4 Connect the blocks and label the signals as shown in the following figure. You might need to resize some of the blocks to accomplish this task.
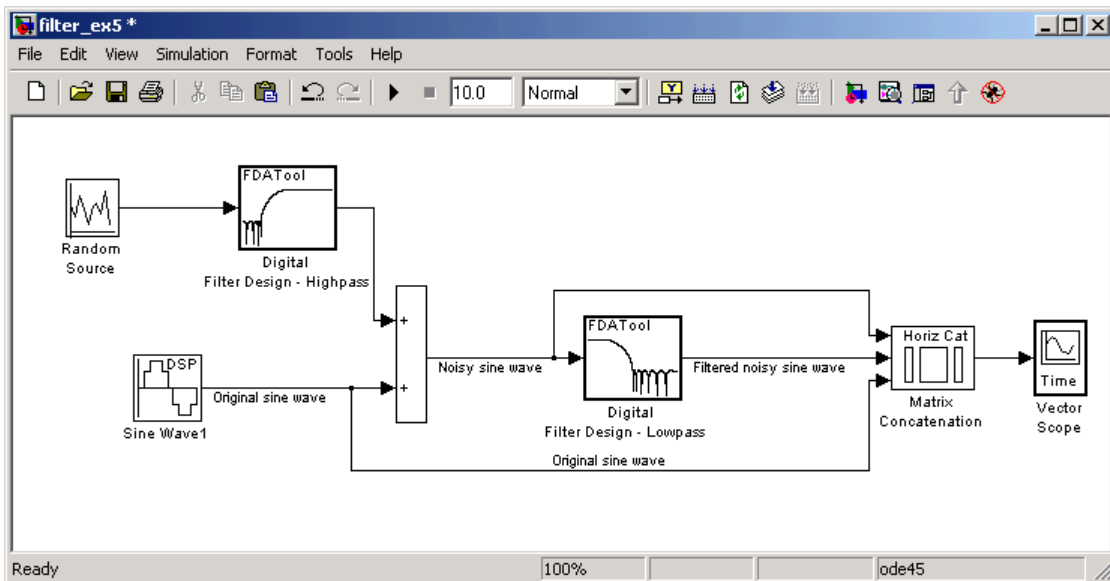
5 From the Simulation menu, select **Configuration Parameters**.

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 Double-click the Vector Scope block and click the **Display Properties** check box. Select the **Channel legend** check box and click **OK**. Next time you run the simulation, a legend appears in the Vector Scope window.

You can also set the color, style, and marker of each channel.

9 In the Vector Scope window, from the **Channels** menu, point to **Ch 1** and set the **Style** to **-**, **Marker** to **None**, and **Color** to **Black**.

Point to **Ch 2** and set the **Style** to **-**, **Marker** to **Diamond**, and **Color** to **Red**.

Point to **Ch 3** and set the **Style** to **None**, **Marker** to *****, and **Color** to **Blue**.
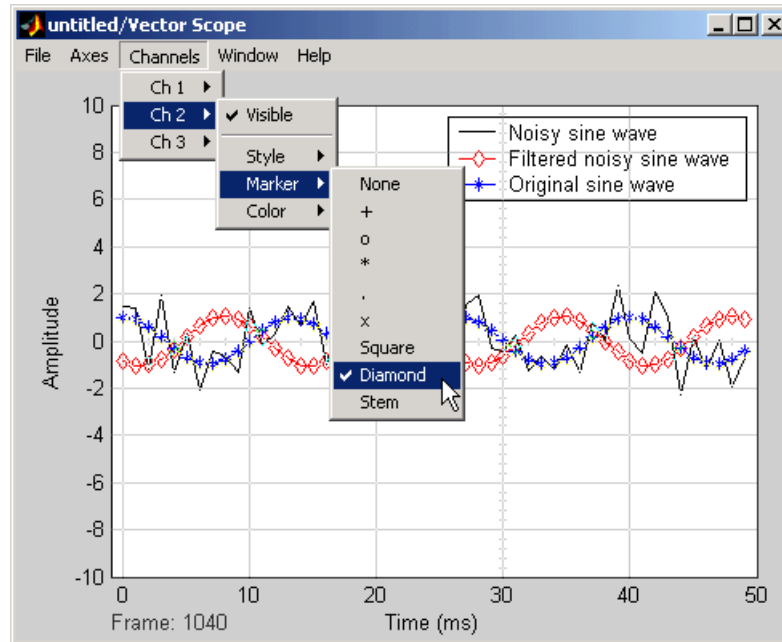
10 Rerun the simulation and compare the original sine wave, noisy sine wave, and filtered noisy sine wave in the **Vector Scope** display.

You can see that the lowpass filter filters out the high-frequency noise in the noisy sine wave.
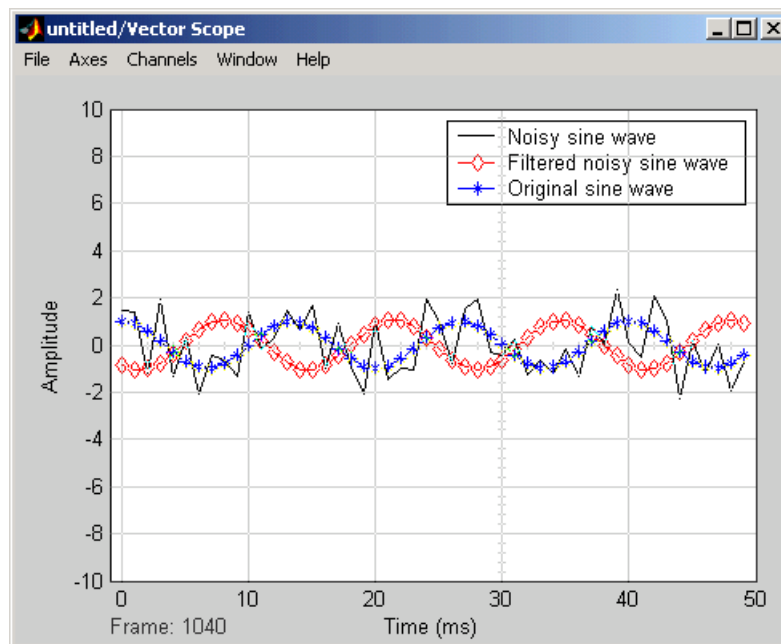
You have now used Digital Filter Design blocks to build a model that removes high frequency noise from a signal. For more information about these blocks, see the Digital Filter Design block reference page. For information on another block capable of designing and implementing filters, see "Filter Realization Wizard" on page 3-32. 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

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 the Signal Processing Blockset.

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 3-20.

This section includes the following topics:

- "Designing and Implementing a Fixed-Point Filter" on page 3-32 — Create a fixed-point filter with the Filter Realization Wizard

- "Setting the Filter Structure and Number of Filter Sections" on page 3-47 — Learn how to change the filter structure and the number of second-order sections in the filter

- "Optimizing the Filter Structure" on page 3-49 — Optimize your filter structure for zero, unity, and negative gains

Alternatively, you can use other MathWorks products, such as the Signal Processing Toolbox and Filter Design Toolbox, to design your filters. Once you design a filter using either toolbox, you can use one of the Signal Processing Blockset's filter implementation blocks, 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

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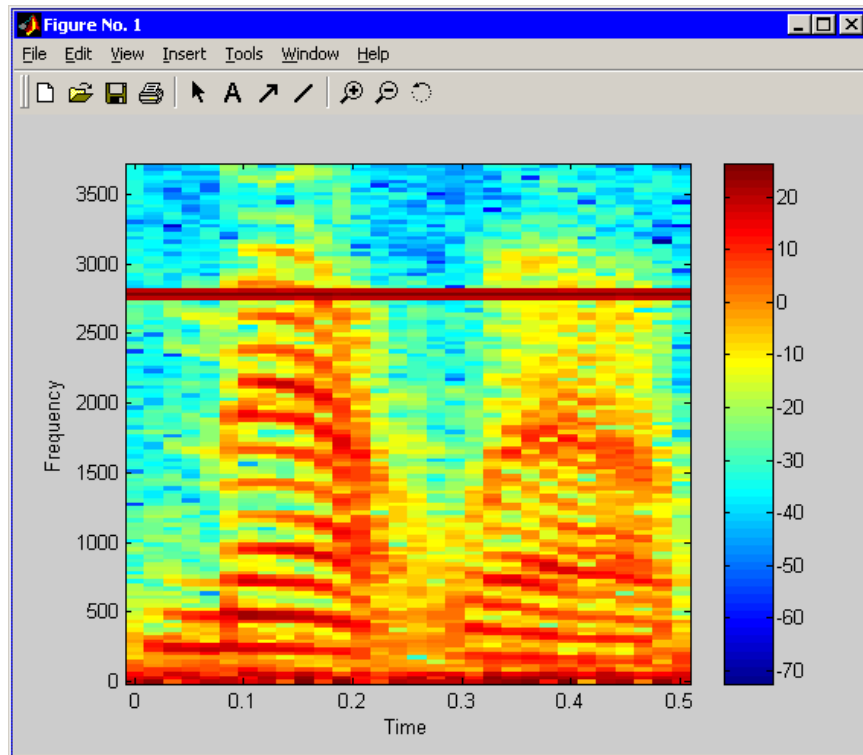
In the spectrogram, you can see the noise signal as a horizontal line at about 2800 Hz, which is equal to 3*Fs/8.

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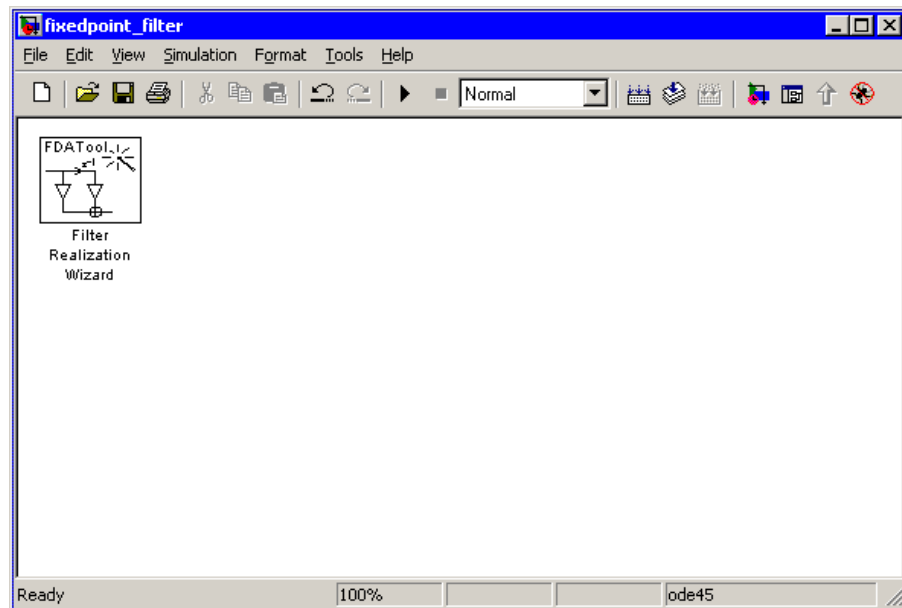
Note You do not have to place a Filter Realization Wizard block in a model in order to use it. You can open the GUI from within a library. However, for purposes of this tutorial, we will keep the Filter Realization Wizard block in the model.

7 Double-click the Filter Realization Wizard block in your model. The **Realize Model** panel of the Filter Design and Analysis Tool (FDATool) appears.
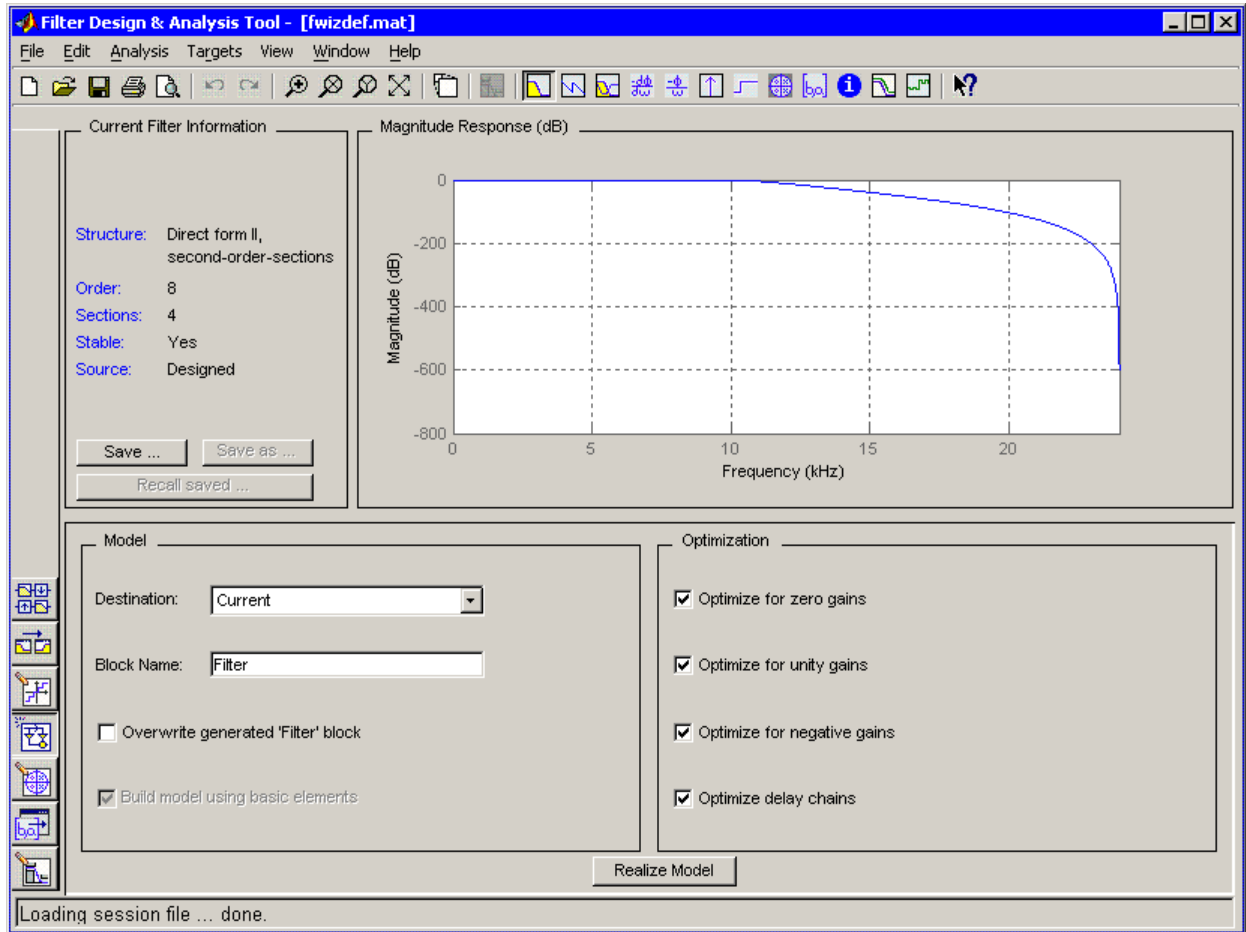
8 Click the Design Filter button on the bottom left of FDATool. This brings forward the **Design Filter** panel of the tool.
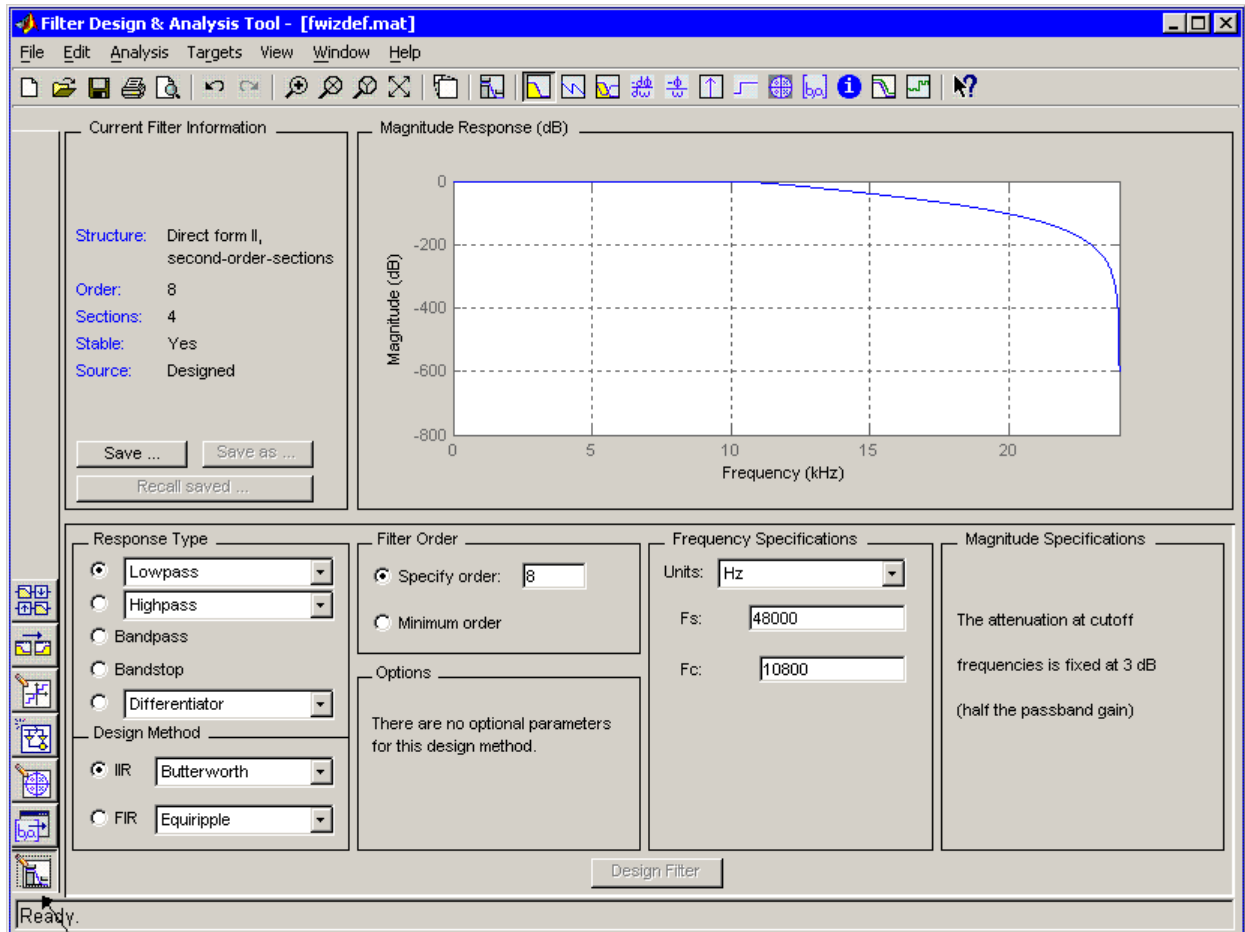
Design Filter button

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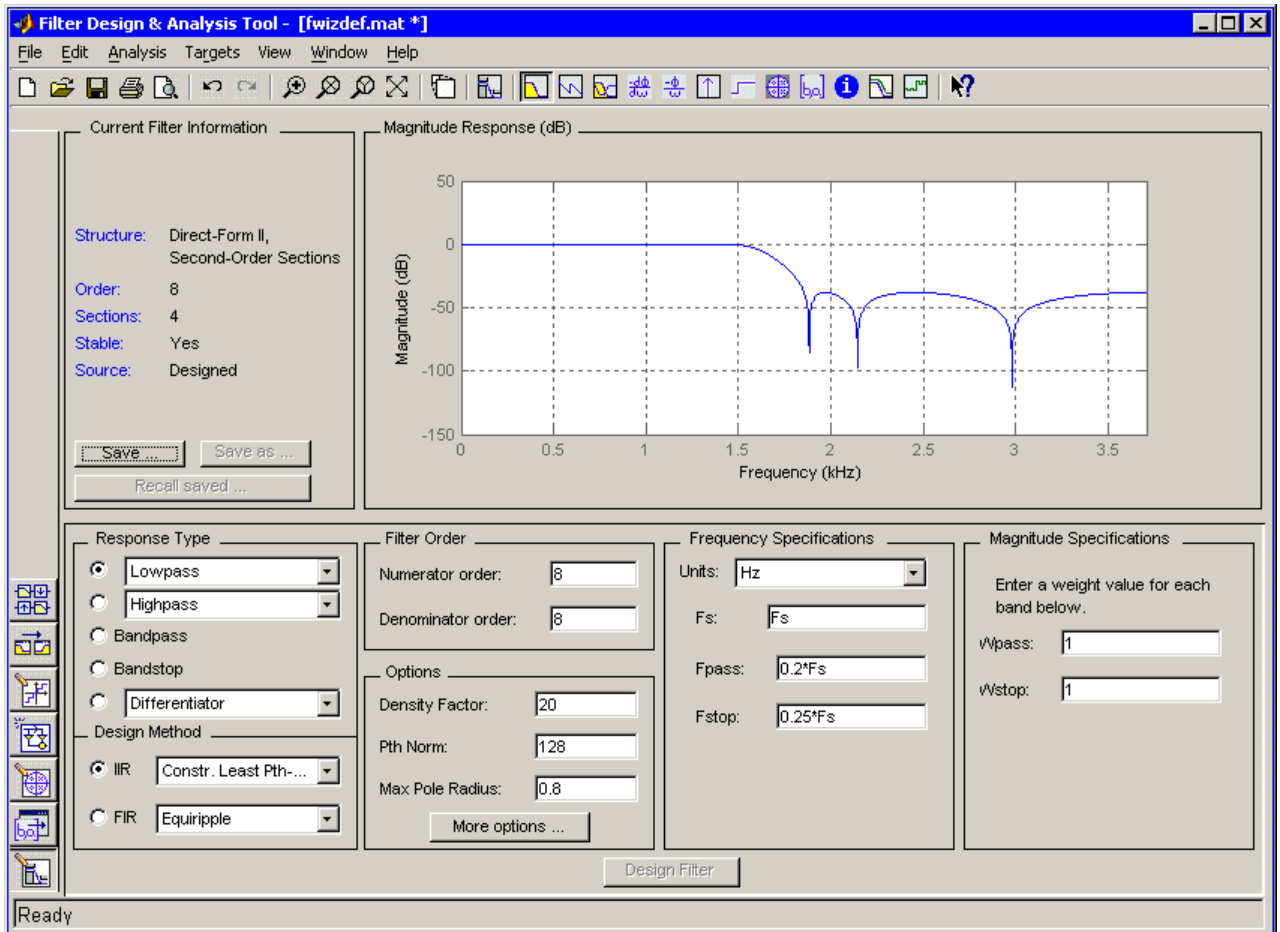
10 Click the **Set Quantization Parameters** button on the bottom left of FDATool. This brings forward the **Set Quantization Parameters** panel of the tool.
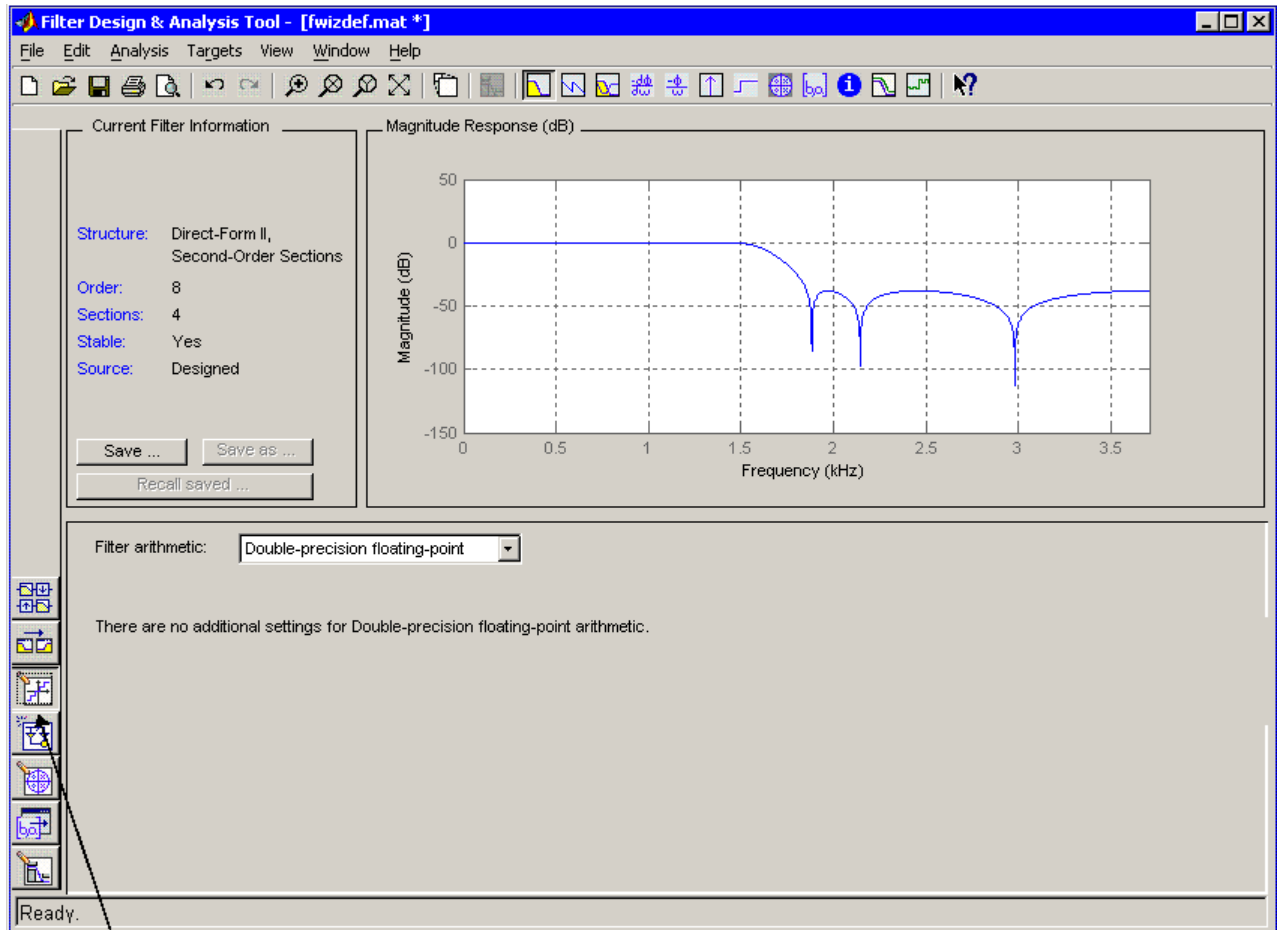
Set Quantization Parameters button

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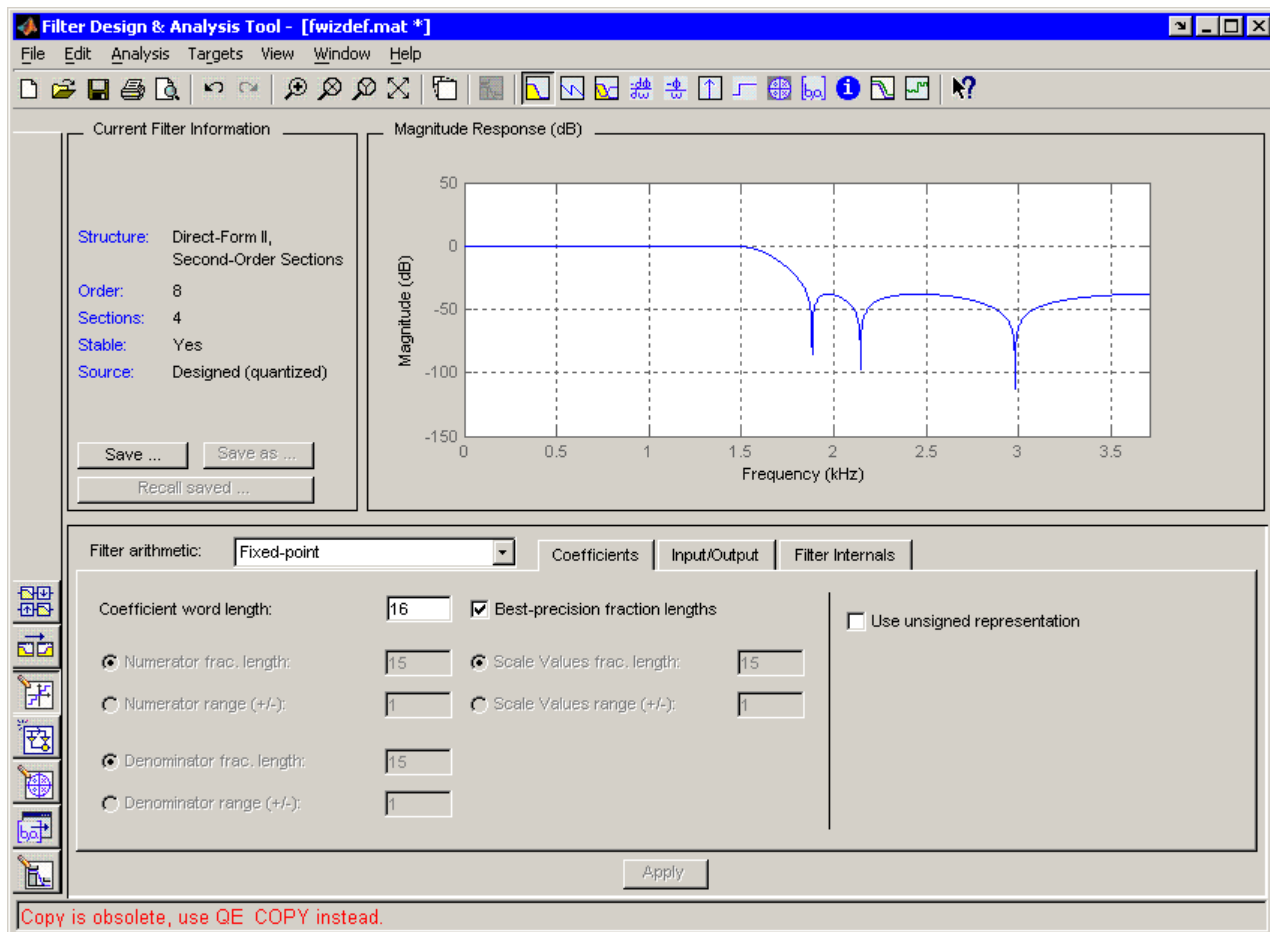
12 Click the Realize Model button on the left side of FDATool. This brings forward the **Realize Model** panel of the tool.
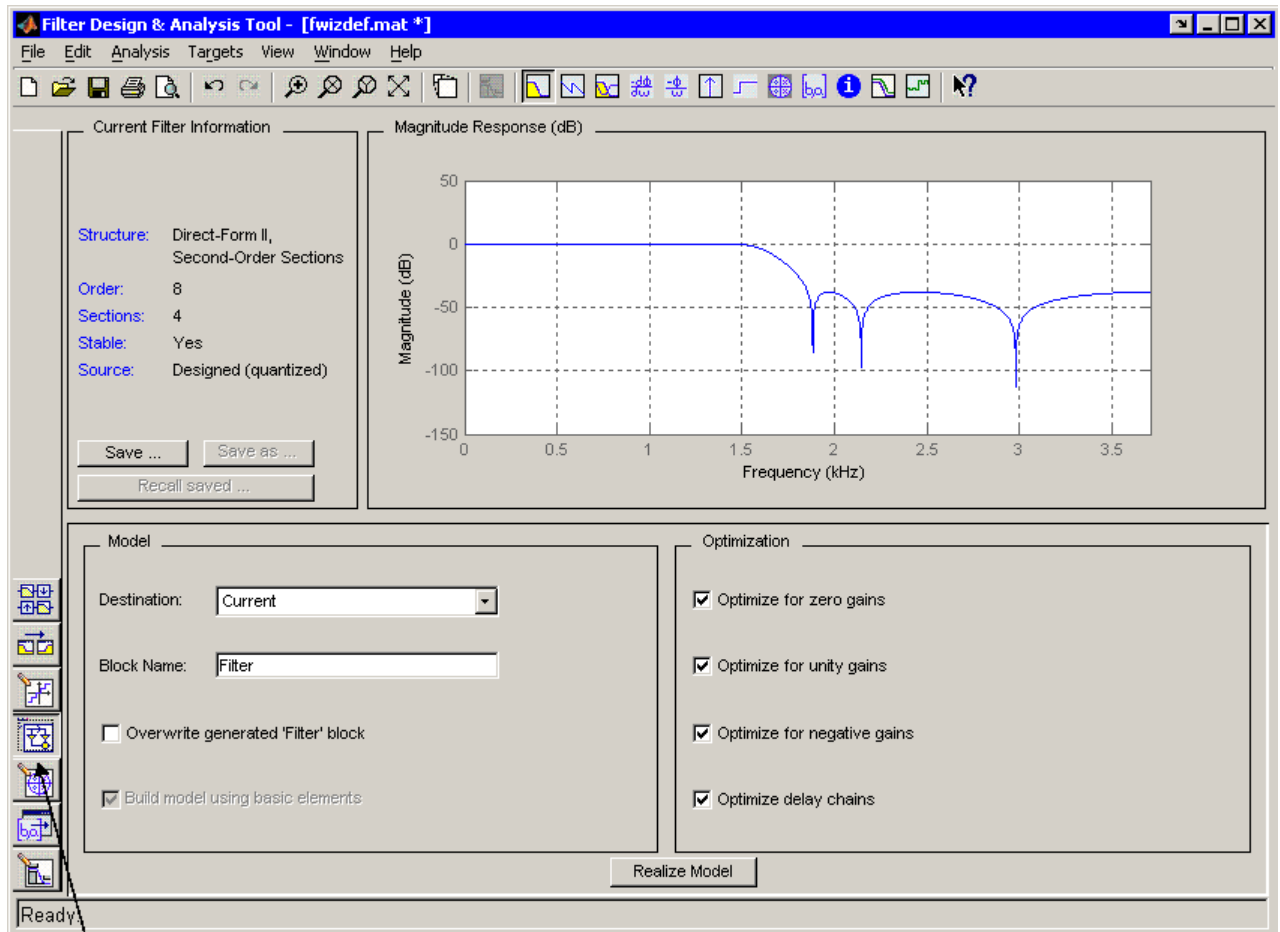
Realize Model button

13 Click the **Realize Model** button on the bottom of FDATool. A block for the new filter appears in your model.
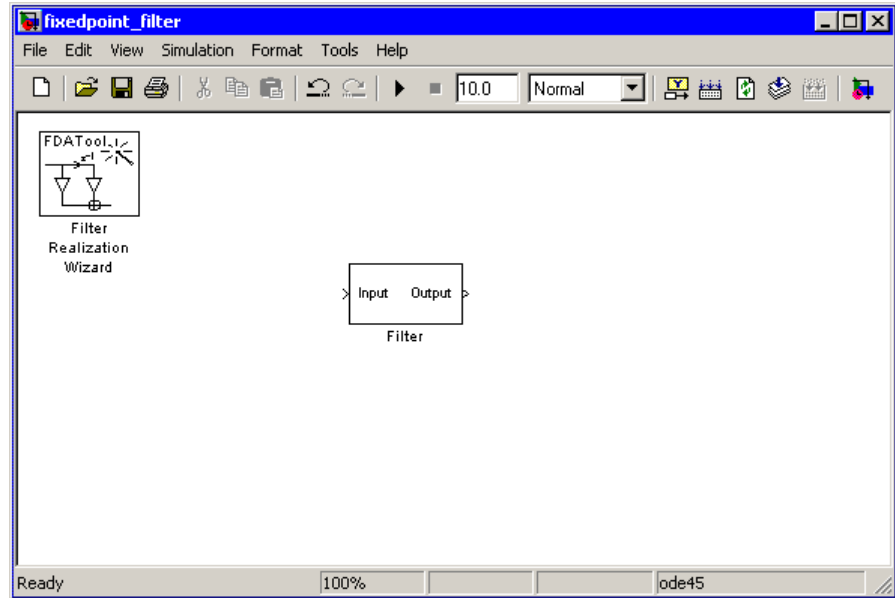
Note You do not have to keep the Filter Realization Wizard block in the same model as your Filter block. However, for this tutorial, we will keep the blocks in the same model.

14 Double-click the Filter block in your model. This will bring up the realization of the filter being represented by the block.
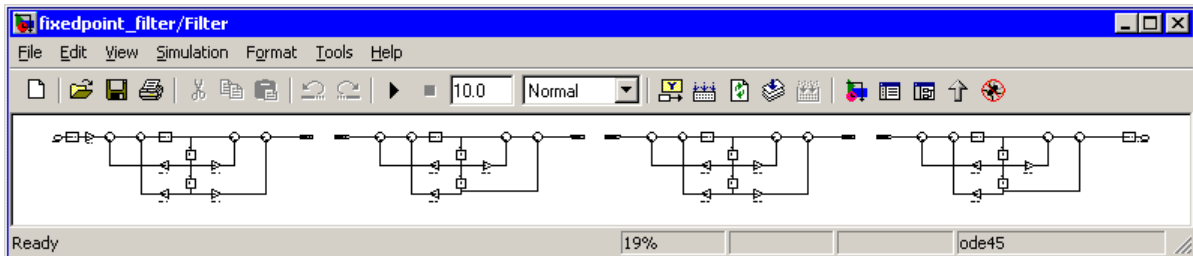
Part 3 — Building a Model to Filter a Signal

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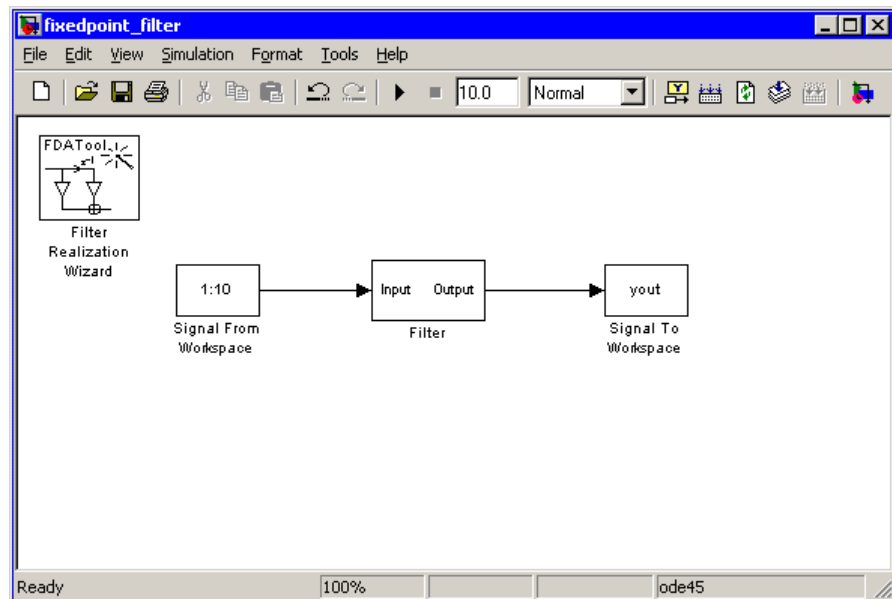
17 Change the **Signal** parameter of the Signal From Workspace block to u by double-clicking on the block.
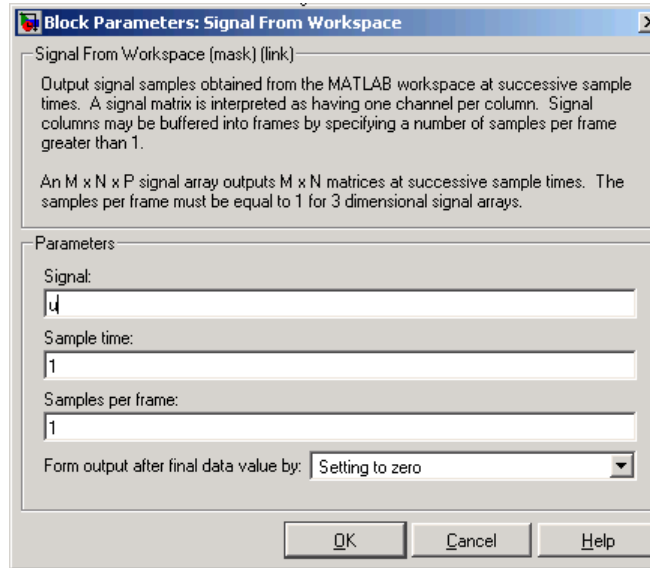
18 Click the **OK** button.

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.
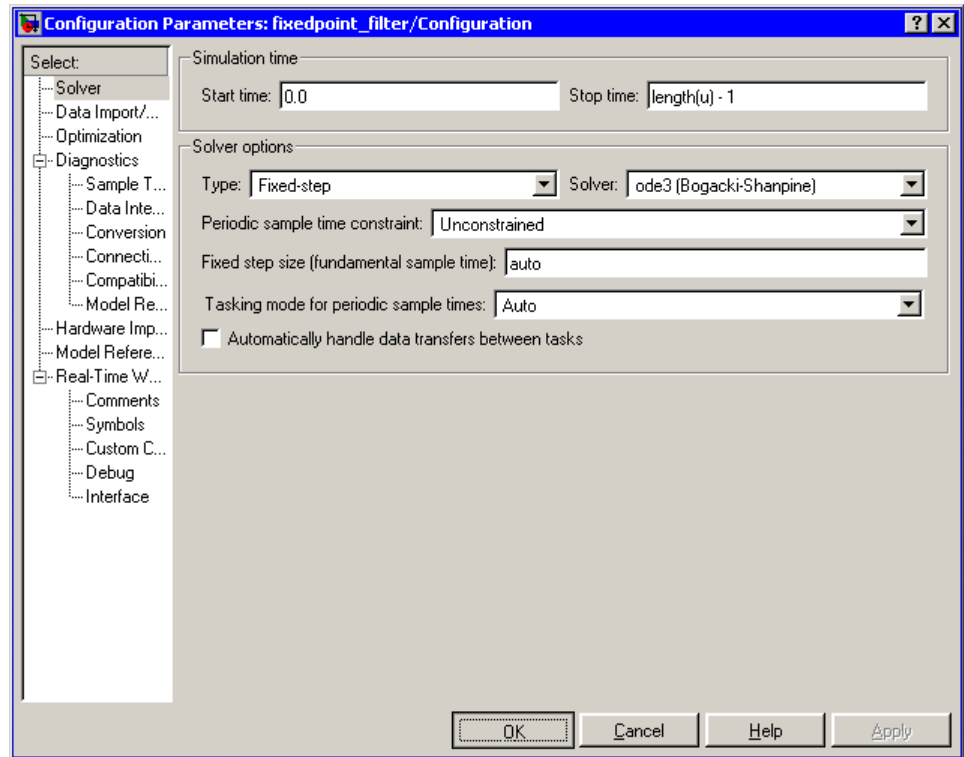
20 Click the **OK** button.

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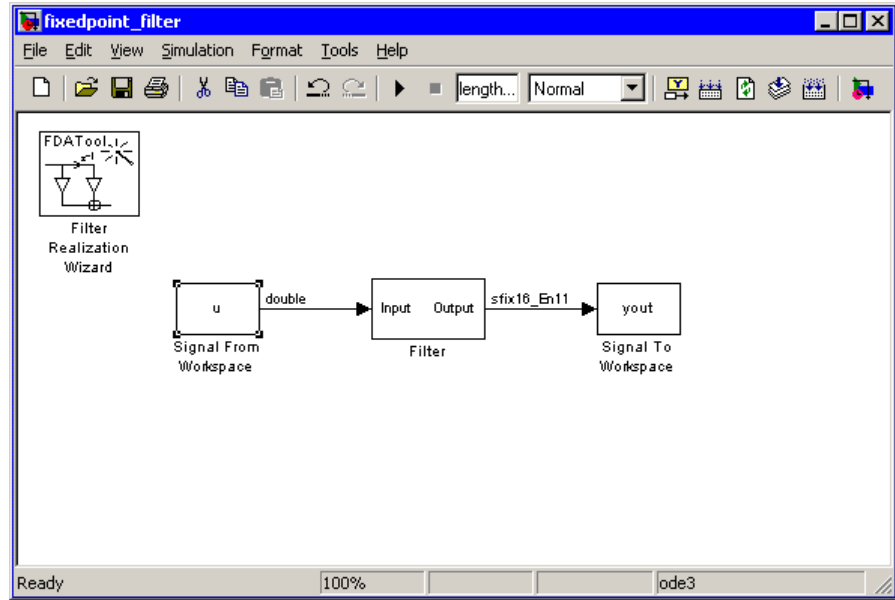
22 Select **Port/Signal Displays** > **Port Data Types** from the **Format** menu. You can you see that a signal of type double is entering your Filter block, and a signal of type sfix16_En11 is exiting your Filter block.

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. You can compare the input and output signals side-by-side.
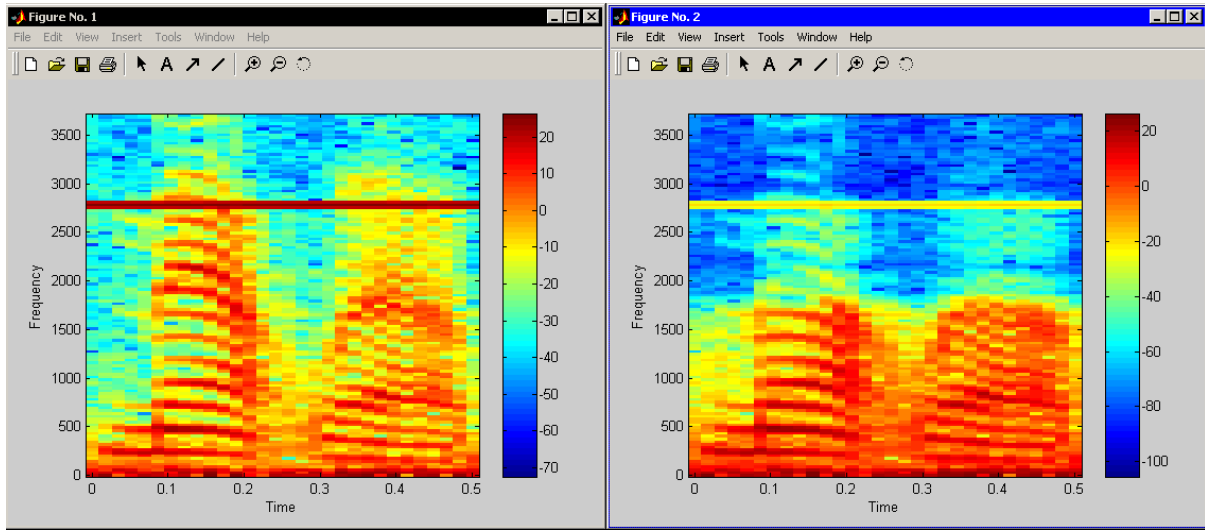
From the colorbars at the side of each spectrogram, you can see that the noise has been reduced by about 40 dB.

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.
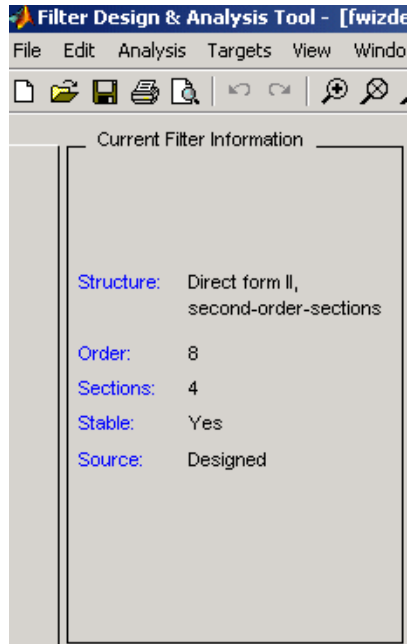
Change the filter structure and number of filter sections of your filter as follows:

- 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.
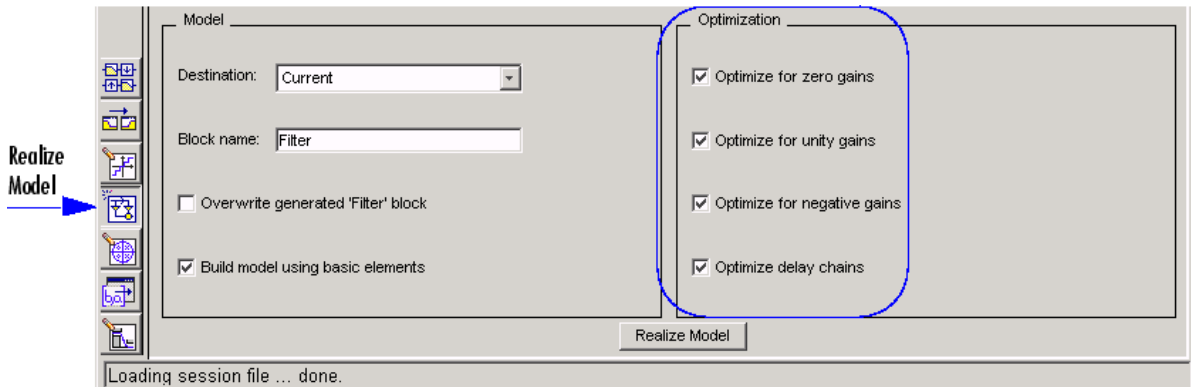
- **Optimize for zero gains** — Remove zero-gain paths.
- **Optimize for unity gains** — Substitute gains equal to one with a wire (short circuit).
- **Optimize for negative gains** — Substitute gains equal to -1 with a wire (short circuit), and change the corresponding sums to subtractions.
- **Optimize 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.
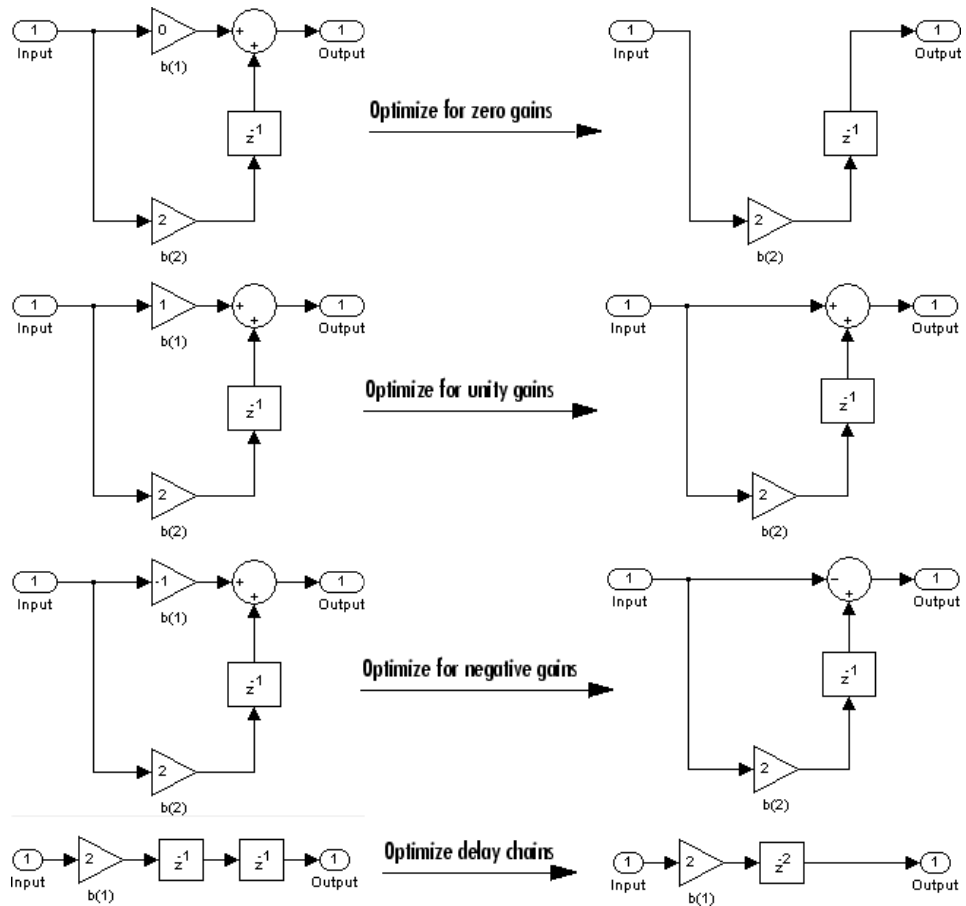
Analog Filter Design Block

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 the Signal Processing Toolbox. For more information on the filter design algorithms, see "Filter Designs" 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

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.

This section includes the following topics:

- "Creating an Acoustic Environment" on page 3-53 — Build a subsystem that models white noise and colored noise added to an input signal

- "Creating an Adaptive Filter" on page 3-54 — Build an adaptive filter using an LMS Filter block

- "Customizing an Adaptive Filter" on page 3-60 — Modify your adaptive filter and change its parameters during simulation

- "Adaptive Filtering Demos" on page 3-64 — Explore the adaptive filtering demos in the Signal Processing Blockset

Creating an Acoustic Environment

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:

1 At the MATLAB command line, type dspanc.

The Signal Processing Blockset Acoustic Noise Cancellation demo opens.
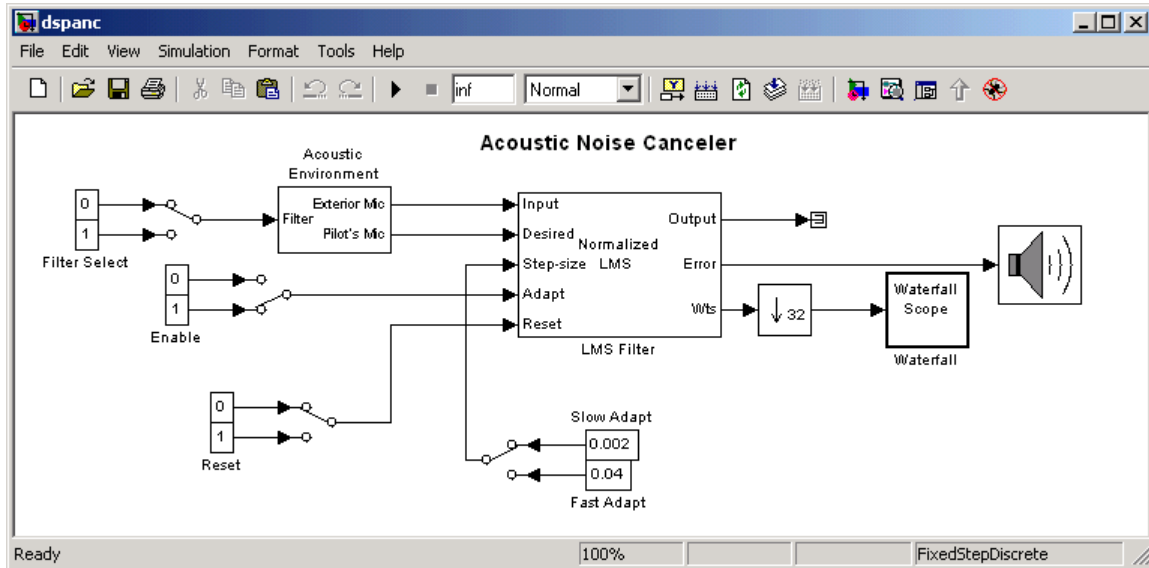
2 Copy and paste the subsystem called Acoustic Environment into a new model file.

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_win32`

   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
To Wave Device	Platform Specific I/O/ Windows	1

3-55

Block	Library	Quantity
Downsample	Signal Operations	1
Waterfall Scope	Signal Processing Sinks	1

5 Connect the blocks so that your model resembles the following figure.
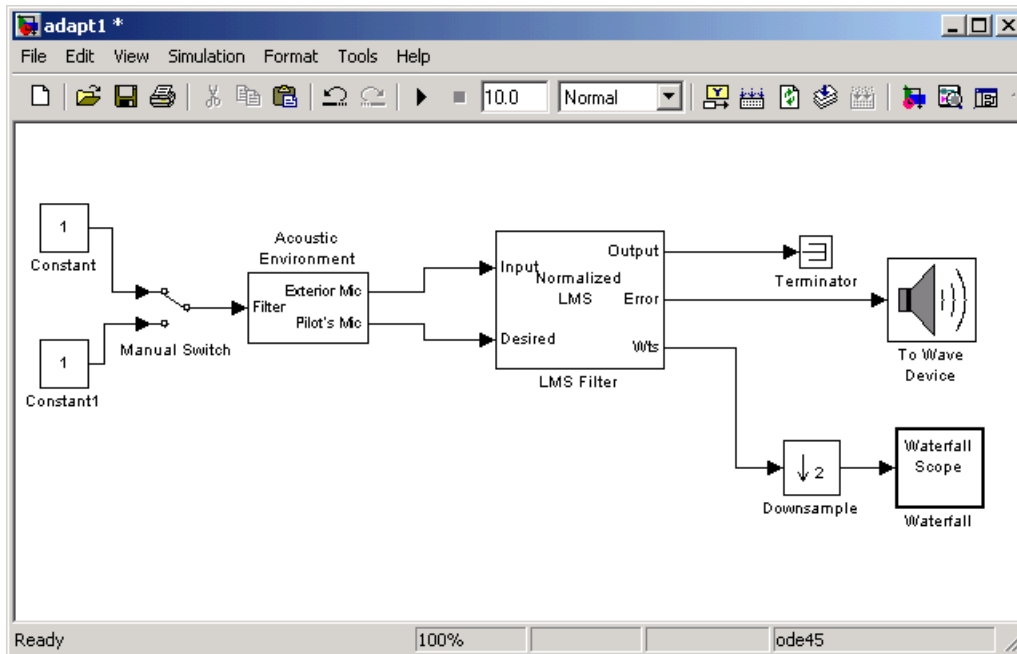
6 Double-click the Constant block. Set the **Constant value** parameter to 0 and then click **OK**.

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

- **Queue duration (seconds)** = 0.4

- **Initial output delay (seconds)** = 0.05

- Select the **Use default audio device** check box.

8 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.

9 Double-click the Waterfall Scope block. The **Waterfall** scope window opens.

10 Click on the Scope parameters button.
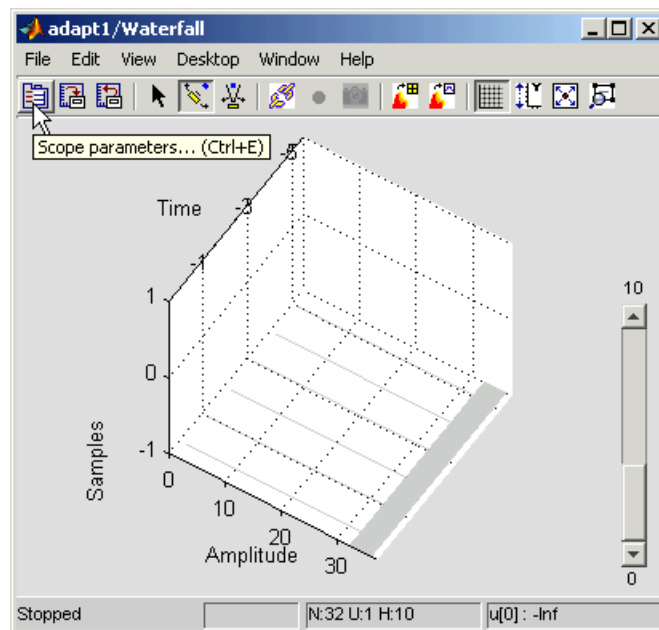
The **Parameters** window opens.
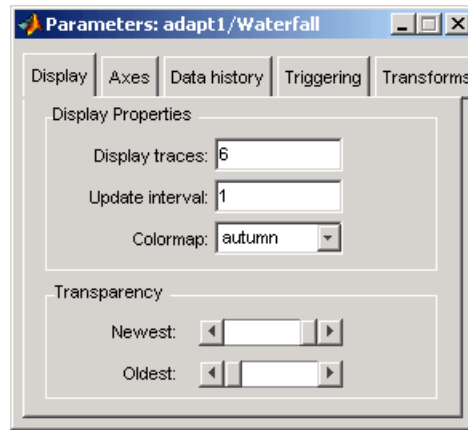
11 Click on the **Axes** tab. Set the parameters as follows:

- **Y Min** = -0.188
- **Y Max** = 0.179

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

- **History traces** = 50
- **Data logging** = All visible

13 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.

14 Click on the **Fit to view** button in the **Waterfall** scope window. Then, click-and-drag the axes until they resemble the following figure.
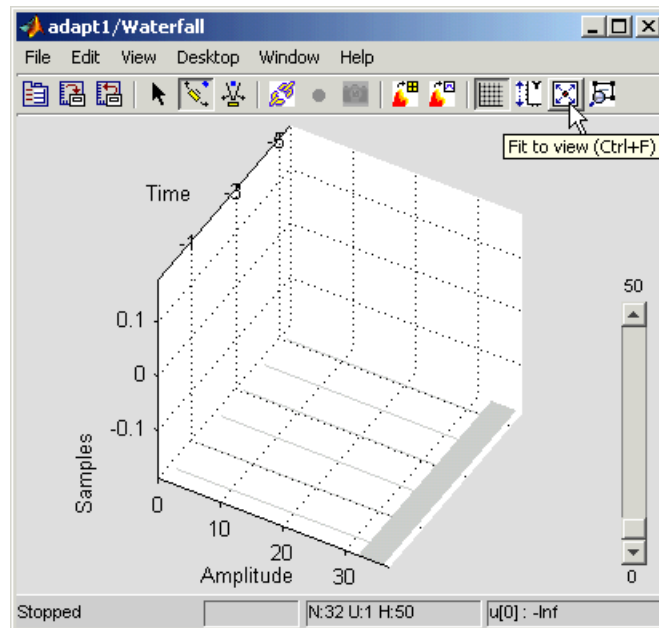
15 In the model window, from the **Simulation** menu, select **Configuration Parameters**. In the **Select** pane, click **Solver**. Set the parameters as follows, and then click **OK**:

- **Stop time** = inf
- **Type** = Fixed-step
- **Solver** = discrete (no continuous states)

16 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.

17 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 3-54, you created an adaptive filter and used it to remove the noise generated by the Acoustic Environment subsystem. In this topic, you modify the adaptive filter and adjust its parameters during simulation. This topic assumes that you are working on a Windows operating system:

1 If the model you created in "Creating an Acoustic Environment" on page 3-53 is not open on your desktop, you can open an equivalent model by typing

    doc_adapt2_win32

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.

Block Parameters: LMS Filter

LMS Filter

Adapts the filter weights based on the chosen algorithm for filtering of the input signal.

Select the Adapt port check box to create an Adapt port on the block. When the input to this port is nonzero, the block continuously updates the filter weights. When the input to this port is zero, the filter weights remain constant.

If the Reset port is enabled and a reset event occurs, the block resets the filter weights to their initial values.

Settings on the "Fixed-point" pane only apply when block inputs are fixed-point signals.

Main | Fixed-point

Parameters

Algorithm: Normalized LMS

Filter length: 40

Specify step size via: Input port

Leakage factor (0 to 1): 1.0

Initial value of filter weights: 0

☑ Adapt port

Reset port: Non-zero sample

☑ Output filter weights

OK | Cancel | Help | Apply

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.
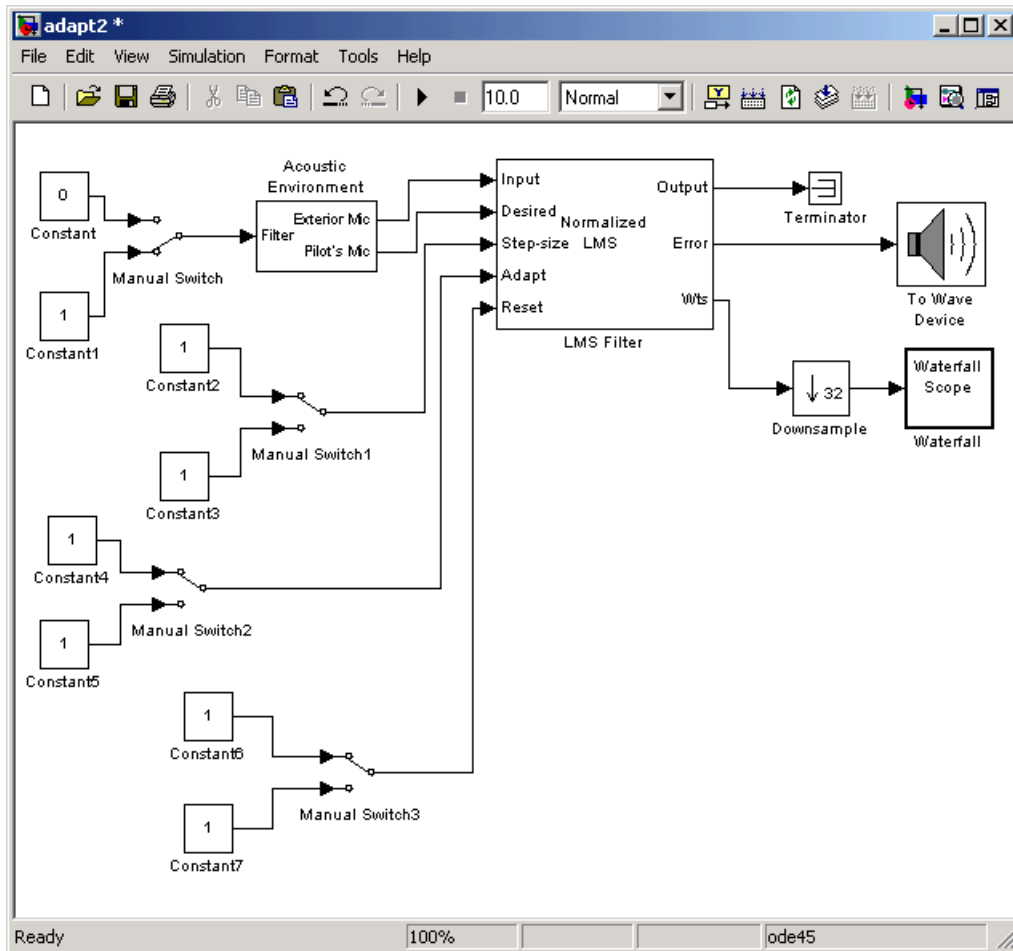
5 Double-click the Constant2 block. Set the block parameters as follows, and then click **OK**:

- **Constant value** = 0.002

- Select the **Interpret vector parameters as 1-D** check box.

- Select the **Show additional parameters** check box.

- **Output data type mode** = Inherit via back propagation

- **Sample time (-1 for inherited)** = inf

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.

- Select the **Show additional parameters** check box.

- **Output data type mode** = `Inherit via back propagation`

- **Sample time (-1 for inherited)** = `inf`

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

The Signal Processing Blockset 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 Linear Prediction	lmsadlp
LMS Adaptive Time-Delay Estimation	lmsadtde

Adaptive Filtering Demos	Commands for Opening Demos in MATLAB
Nonstationary Channel Estimation	kalmnsce
RLS Adaptive Noise Cancellation	rlsdemo

Opening Demos

To open the adaptive filter demos, click on the links in the following 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 dsp at the MATLAB command line.

Multirate Filters

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.

This section includes the following topics:

- "Filter Banks" on page 3-66 — Review of dyadic analysis filter banks and dyadic synthesis filter banks
- "Multirate Filtering Demos" on page 3-74 — Explore the multirate filtering demos in the Signal Processing Blockset

Filter Banks

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.
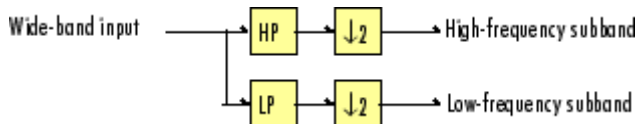
Each unit consists of a lowpass (LP) and highpass (HP) FIR filter pair, followed by a decimation by a factor of 2. The filters are halfband filters with

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*.
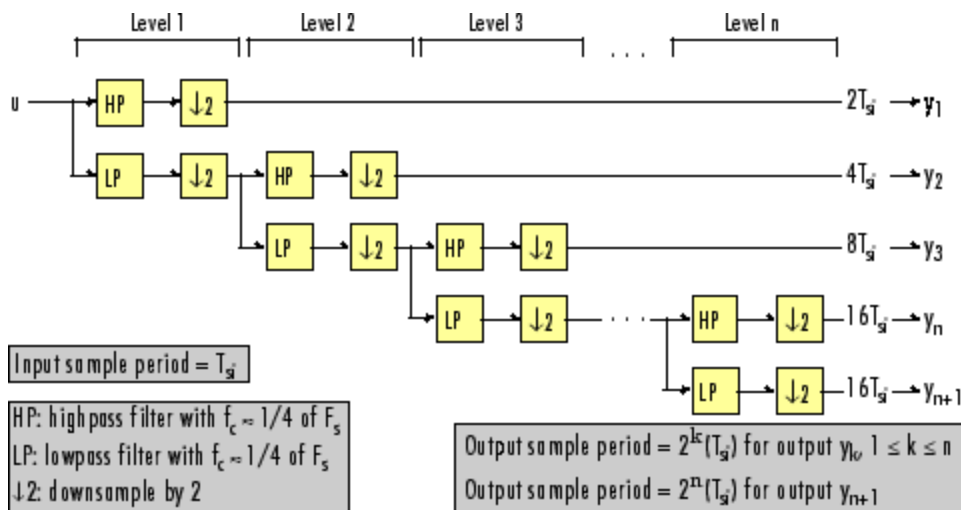
n-Level Asymmetric Dyadic Analysis Filter Bank

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.
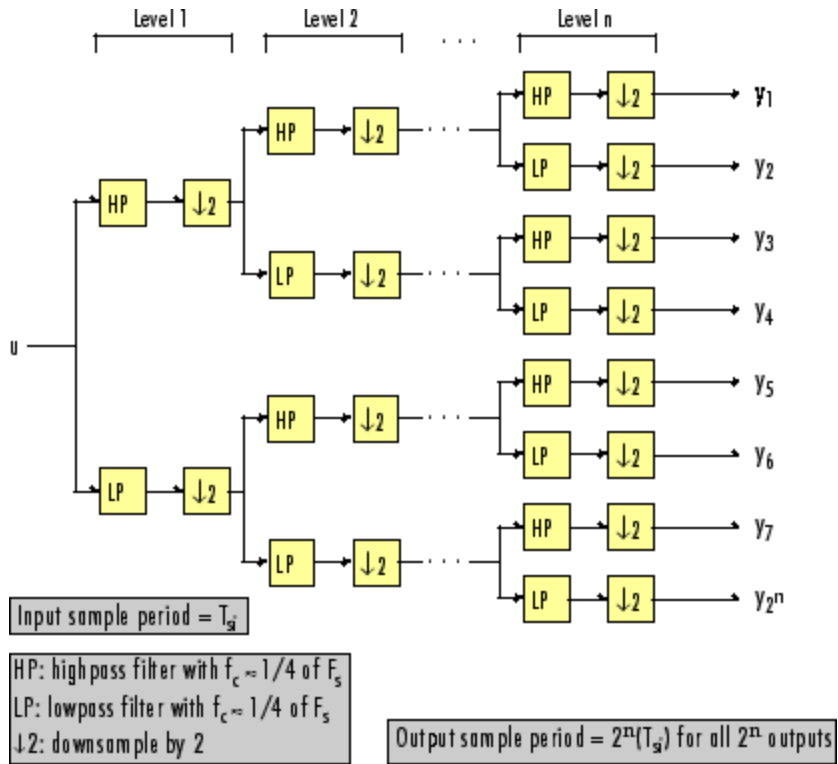
n-Level Symmetric Dyadic Analysis Filter Bank

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

Notable Characteristics of Asymmetric and Symmetric Dyadic Analysis Filter Banks (Continued)

Characteristic	N-Level Symmetric	N-Level Asymmetric
Bandwidth and Number of Samples in Output Subbands	For an input with bandwidth BW and N samples, all outputs have bandwidth $BW / 2^n$ and $N / 2^n$ samples.	For an input with bandwidth BW and N samples, y_k has the bandwidth BW_k, and N_k samples, where $$BW_k = \begin{cases} BW/2^k & (1 \le k \le n) \\ BW/2^n & (k = n + 1) \end{cases}$$ $$N_k = \begin{cases} N/2^k & (1 \le k \le n) \\ N/2^n & (k = n + 1) \end{cases}$$ The bandwidth of, and number of samples in each subband (except the last) is half those of the previous subband. The last two subbands have the same bandwidth and number of samples since they originate from the same level in the filter bank.
Output Sample Period	All output subbands have a sample period of $2^n(T_{si})$	Sample period of kth output $$= \begin{cases} 2^k(T_{si}) & (1 \le k \le n) \\ 2^n(T_{si}) & (k = n + 1) \end{cases}$$ 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.
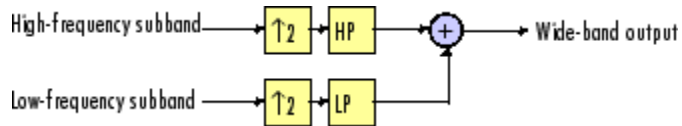
Each unit consists of a lowpass (LP) and highpass (HP) FIR filter pair,
preceded by an interpolation by a factor of 2. The filters are halfband filters
with a cutoff frequency of $F_s / 4$, a quarter of the input sampling frequency.
Each filter passes the frequency band that the other filter stops.

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*.

---
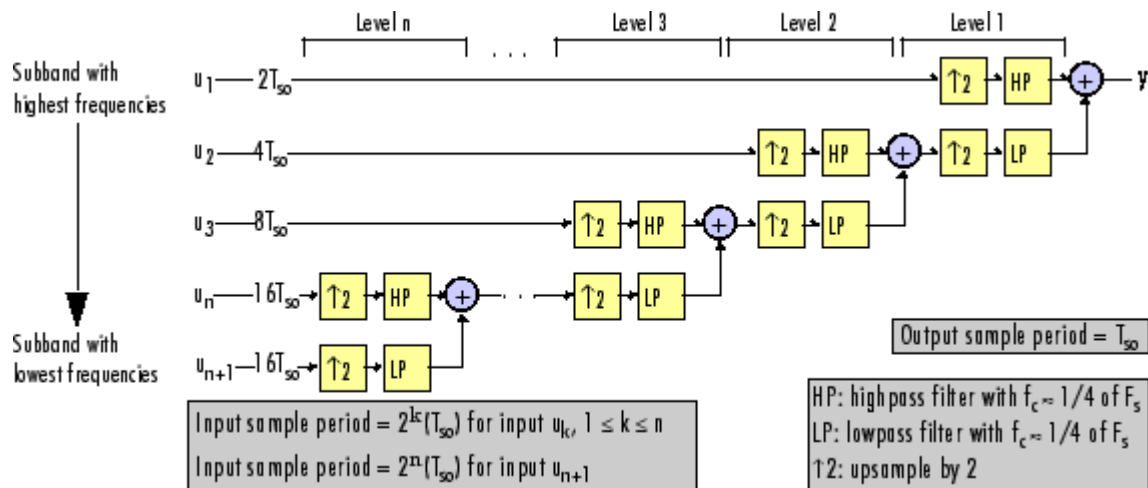
n-Level Asymmetric Dyadic Synthesis Filter Bank

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.
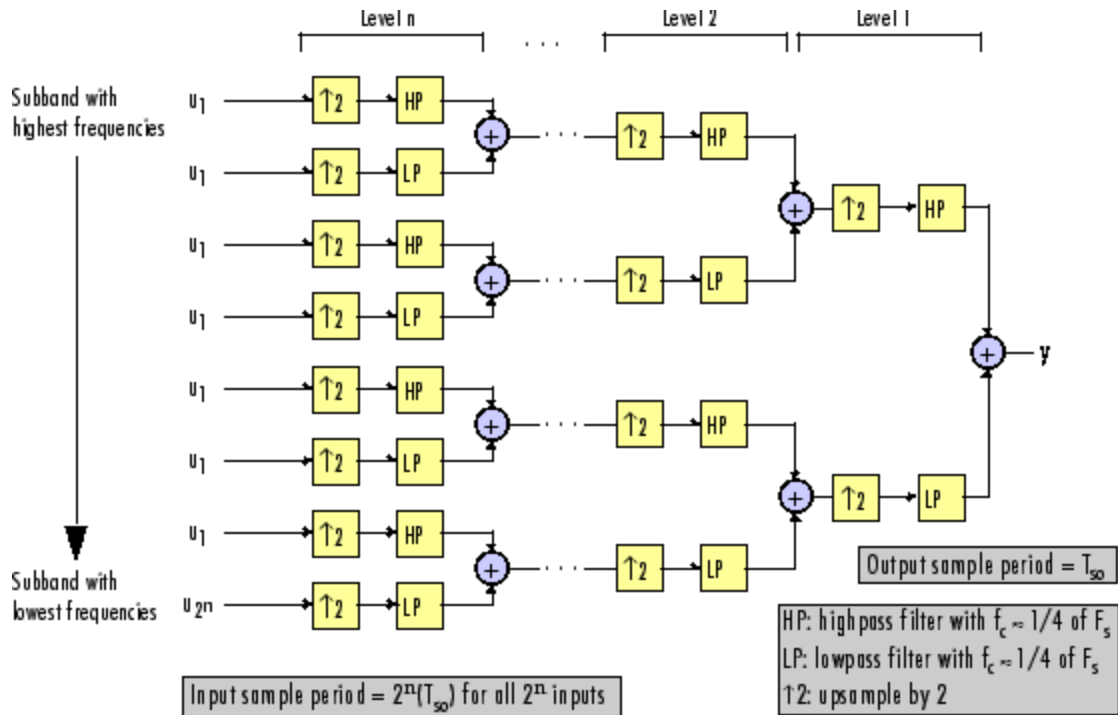
n-Level Symmetric Dyadic Synthesis Filter Bank

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	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.	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.

Notable Characteristics of Asymmetric and Symmetric Dyadic Synthesis Filter Banks (Continued)

Characteristic	N-Level Symmetric	N-Level Asymmetric
Number of Input Subbands	2^n	n+1
Bandwidth and Number of Samples in Input Subbands	All inputs subbands have bandwidth BW / 2^n and N / 2^n samples, where the output has bandwidth BW and N samples.	For an output with bandwidth BW and N samples, the kth input subband has the following bandwidth and number of samples. $$BW_k = \begin{cases} BW/2^k & (1 \le k \le n) \\ BW/2^n & (k = n+1) \end{cases}$$ $$N_k = \begin{cases} N/2^k & (1 \le k \le n) \\ N/2^n & (k = n+1) \end{cases}$$
Input Sample Periods	All input subbands have a sample period of $2^n(T_{so})$, where the output sample period is T_{so}.	Sample period of kth input subband $$= \begin{cases} 2^k(T_{so}) & (1 \le k \le n) \\ 2^n(T_{so}) & (k = n+1) \end{cases}$$ where the output sample period is T_{so}.
Total Number of Input Samples	The number of samples in the output is always equal to the total number of samples in all of the input subbands.	
Wavelet Applications	In wavelet applications, the highpass and lowpass wavelet-based filters are carefully selected so that the aliasing introduced by the decimation in the dyadic *analysis* filter bank is exactly canceled in the reconstruction of the signal in the dyadic *synthesis* filter bank.	

For more information, see Dyadic Synthesis Filter Bank.

Multirate Filtering Demos

The Signal Processing Blockset provides a collection of multirate filtering demos that illustrate typical applications of the multirate filtering blocks, listed in the following table.

Multirate Filtering Demos	Commands for Opening Demos in MATLAB
Denoising	dspwdnois
Interpolation of a Sinusoidal Signal	dspintrp
Multistage Multirate Filtering Suite	dspmrf_menu
Sample Rate Conversion	dspsrcnv
Sigma-Delta A/D Converter	dspsdadc
Three-Channel Wavelet Transmultiplexer	dspwvtrnsmx
Wavelet Perfect Reconstruction Filter Bank	dspwpr
Wavelet Reconstruction	dspwlet

Opening Demos

To open the multirate filter demos, click on the links in the following 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 dsp at the MATLAB command line.

4

Transforms

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

Signals in the Time Domain

You can use the Signal Processing Blockset 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

This section includes the following topics:

- "Displaying Time-Domain Data" on page 4-2 — Use the Vector Scope block to display two frame-based signals in the time domain
- "Transforming Time-Domain Data into the Frequency Domain" on page 4-5 — Use the FFT block to transform two, frame-based sinusoidal signals from the time domain to the frequency domain

Displaying Time-Domain Data

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.
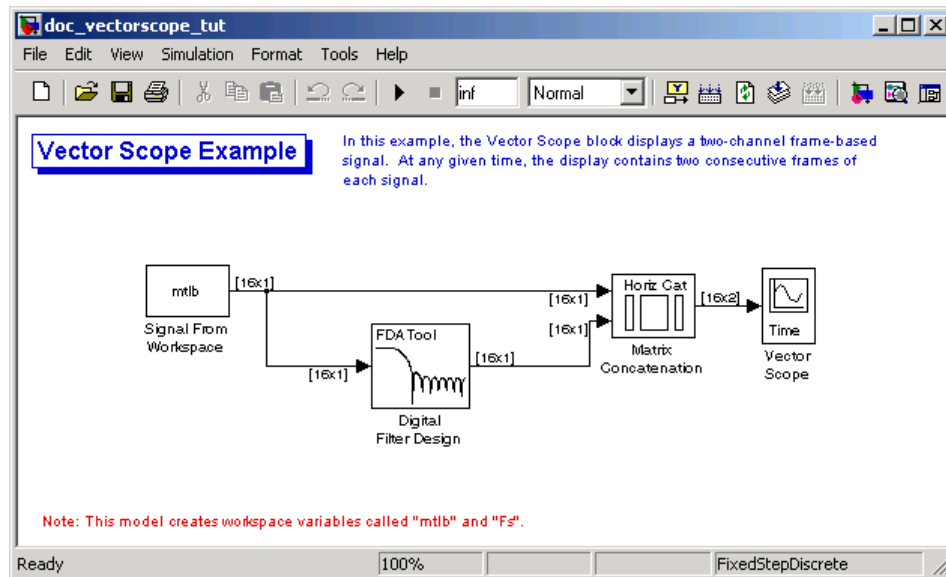
When you run this model, two frame-based signals are displayed in the vectorscope_tut/Vector Scope window.

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 signal's frame period 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 Concatenation block. The **Block Parameters: Matrix Concatenation** dialog box opens.

14 Set the block parameters as follows:

- **Number of inputs** = 2
- **Concatenation method** = Horizontal.

Based on these parameters, the Matrix Concatenation 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 original and filtered signal appear in the Vector Scope window. You have now successfully displayed two frame-based signals in the time domain using the Vector Scope block.

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.
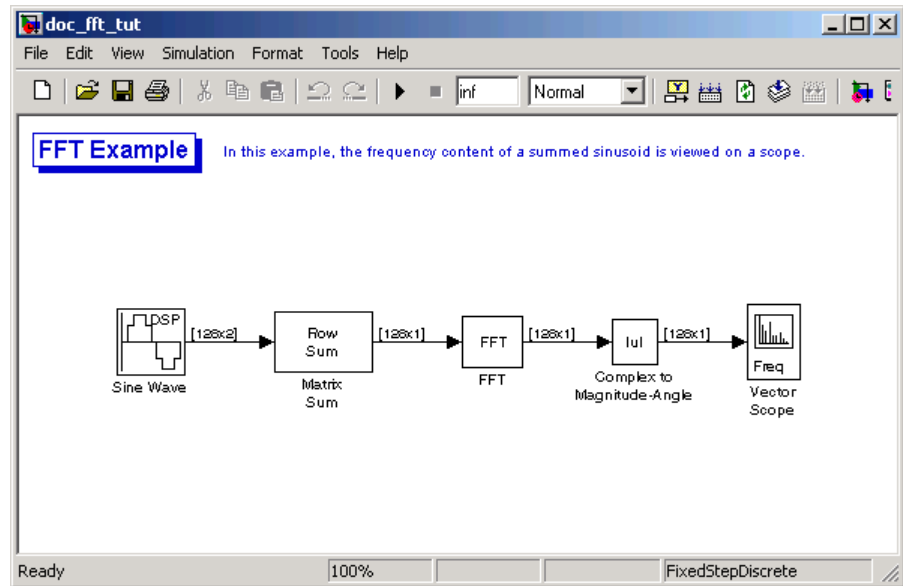
2 Double-click the Sine Wave block. The **Block Parameters: Sine Wave** dialog box opens.

3 Set the block parameters as follows:

- **Amplitude** = 1

- **Frequency =** [15 40]

- **Phase offset =**0 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 along** parameter to Rows, and then click **OK**.

Since 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 0.015 and 0.04 kHz, as expected.
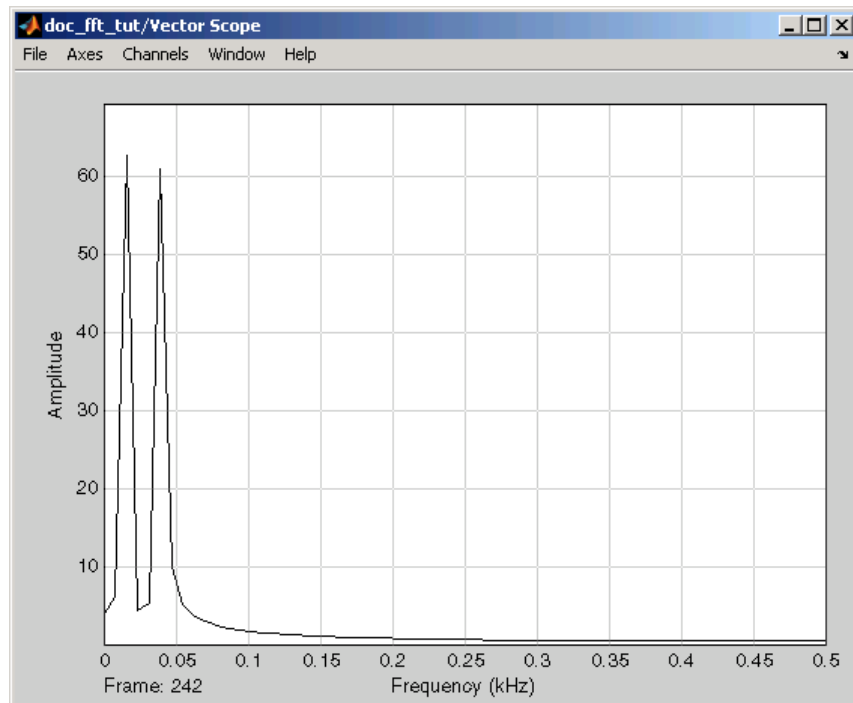
You have now transformed two, frame-based sinusoidal signals from the time domain to the frequency domain.

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.

Signals in the Frequency-Domain

You can use the Signal Processing Blockset 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.

This section includes the following topics:

- "Displaying Frequency-Domain Data" on page 4-9 — Use the Spectrum Scope block to display two, frame-based signals in the frequency domain

- "Transforming Frequency-Domain Data into the Time Domain" on page 4-13 — Use the IFFT block to transform two, frame-based sinusoidal signals from the frequency domain to the time domain

Displaying Frequency-Domain Data

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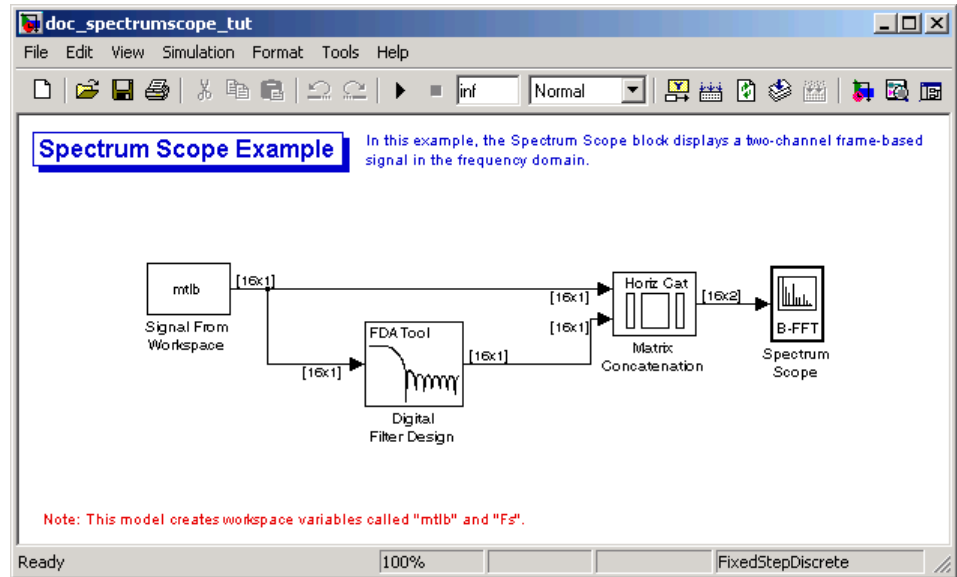
Also, the variables Fs and mtlb are loaded into the MATLAB workspace.

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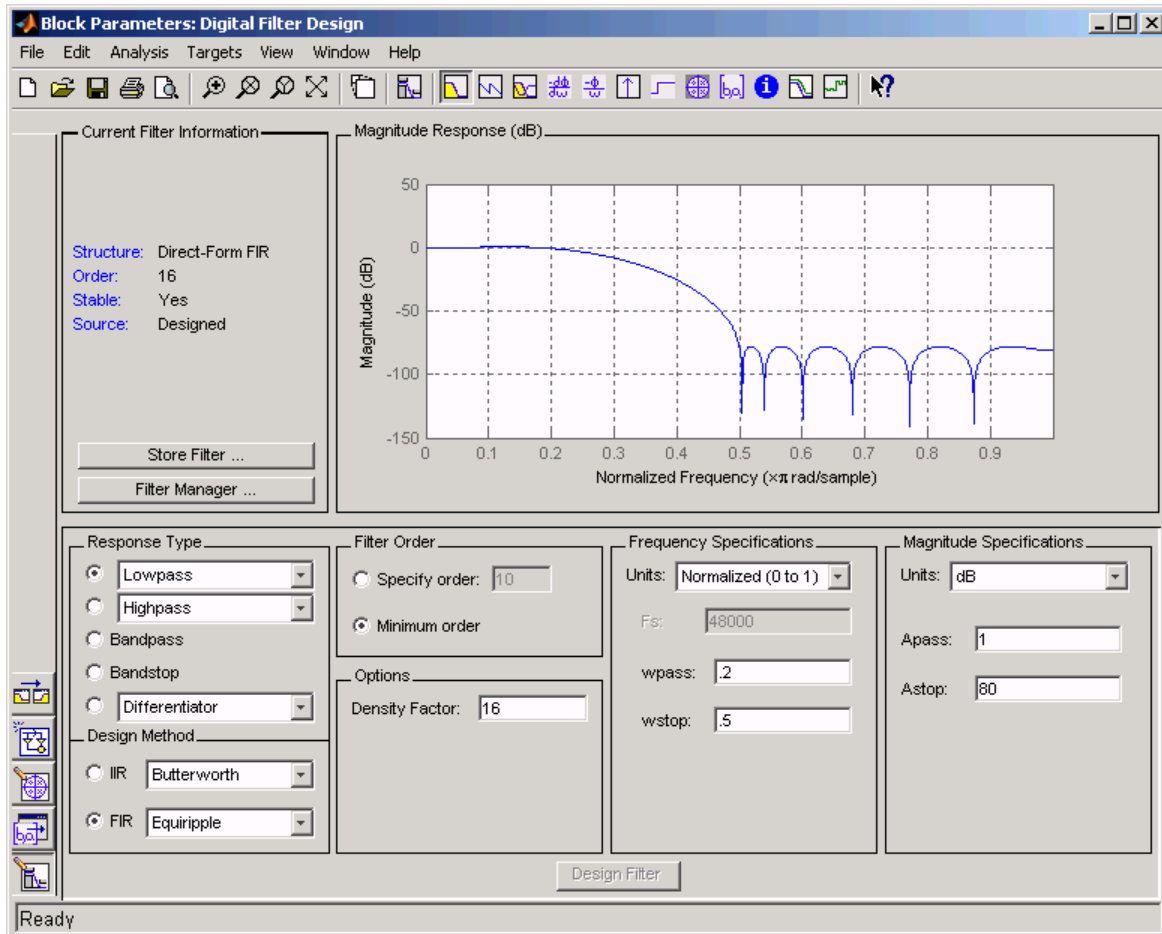
4 Double-click the Matrix Concatenation block. Set the block parameters as follows, and then click **OK**:

- **Number of inputs** = 2

- **Concatenation method** = Horizontal

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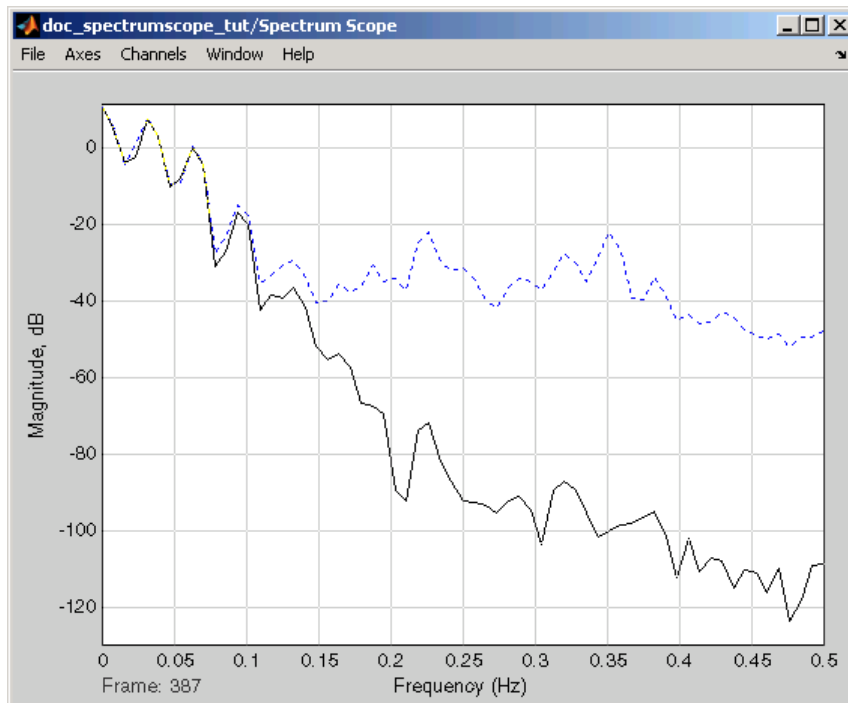
The FFT of the first input signal, from column one, is the blue dotted line. The FFT of the second input signal, from column two, is the black solid line. Every time the scope display is updated, 128 points are plotted for each channel.

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

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.
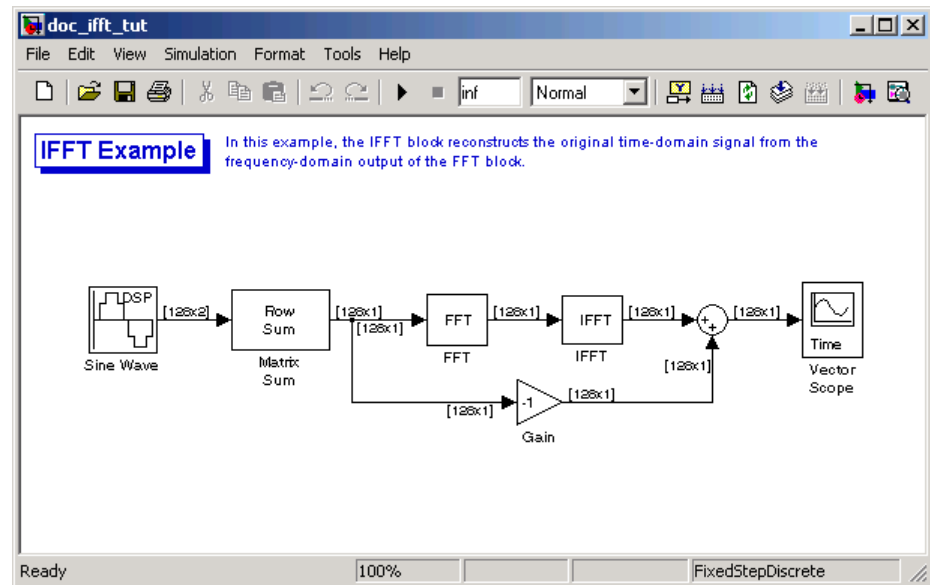
2 Double-click the Sine Wave block. The **Block Parameters: Sine Wave** dialog box opens.

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 along** parameter to Rows, and then click **OK**.

Since 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 IIFT 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.
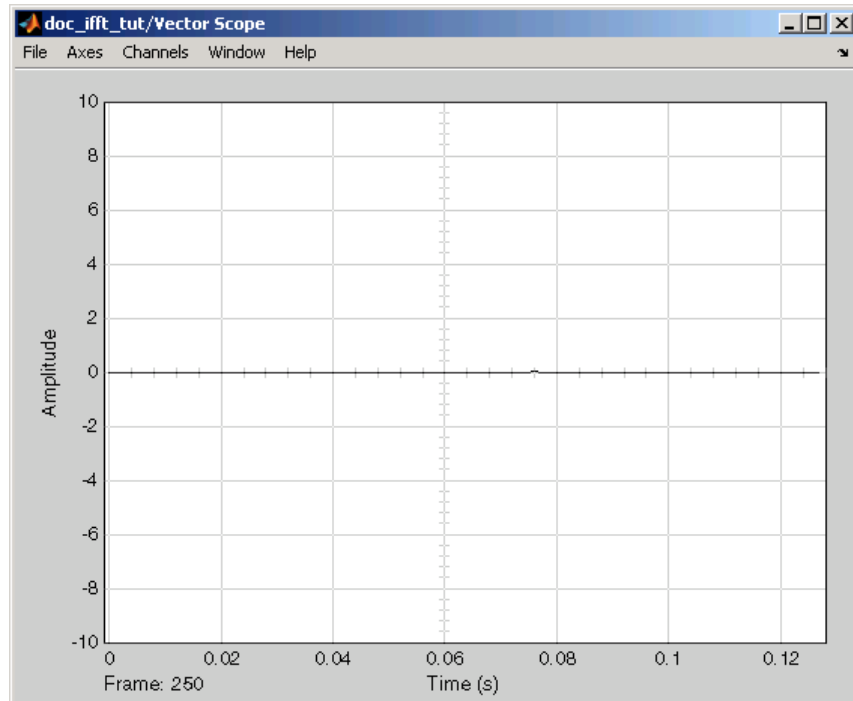
The flat line on the scope suggests that there is no difference between the original signal and the estimate of the original signal. Therefore, the IFFT block has accurately reconstructed the original time-domain signal from the frequency-domain input.

14 Right-click in the Vector Scope window, and select **Autoscale**.
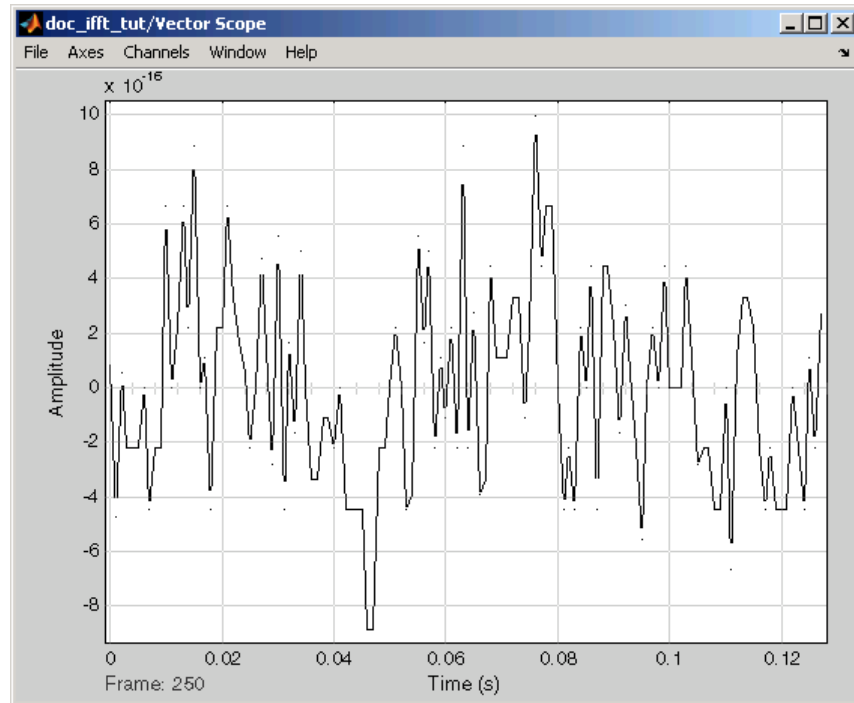
In actuality, the two signals are identical to within round-off error. The previous figure shows the enlarged trace. The differences between the two signals is on the order of 10^{-15}.

Linear and Bit-Reversed Output Order

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 butterfly operation, in some situations, the FFT block runs 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.

This section includes the following topic:

- "Finding the Bit-Reversed Order of Your Frequency Indices" on page 4-18 — Transform linearly ordered frequency indices into bit-reversed frequency indices

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.
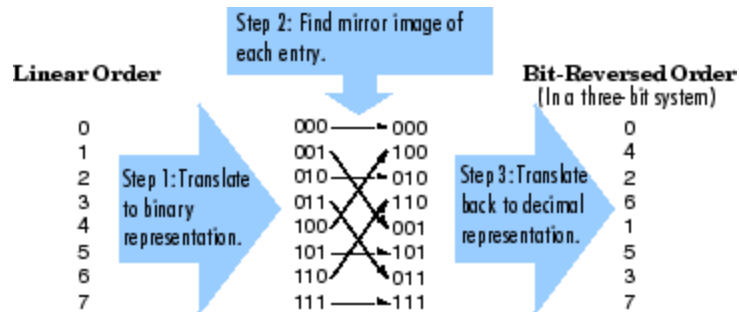
The next diagram illustrates the linear and bit-reversed outputs of the FFT block. The output values are the same, but they appear in different order.
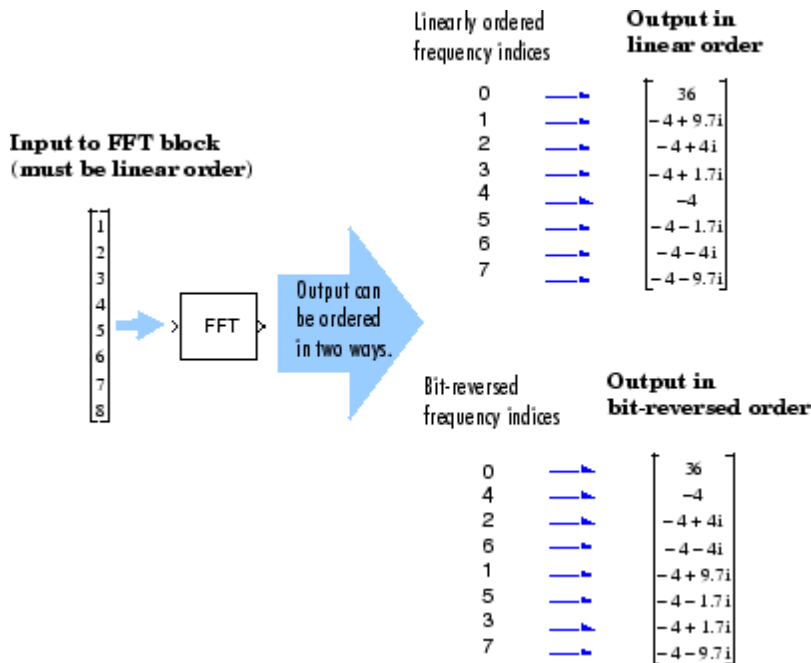
Quantizers

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

You can use blocks from the Signal Processing Blockset Quantizers library to design scalar quantizer encoders and decoders. 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 would be distorted and the speech quality would degrade. In this section, you create two scalar quantizer encoders and two scalar quantizer decoders and use them to encode and decode signals in a demo model.

This section includes the following topics:

- "Analysis and Synthesis of Speech" on page 5-2 — Learn the theory behind signal transmission

- "Identifying Your Residual Signal and Reflection Coefficients" on page 5-4 — Define the residual signal and the reflection coefficients in your MATLAB workspace

- "Creating a Scalar Quantizer" on page 5-5 — Design two scalar quantizer encoders and two scalar quantizer decoders and use them to quantize your residual signal and reflection coefficients

Analysis and Synthesis of Speech

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.

In narrowband digital speech compression, speech signals are sampled at a rate of 8000 samples per second. Typically, each sample is 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. An implementation of this compression technique is presented in the linear prediction coefficient (LPC) Analysis and Synthesis of Speech (`dsplpc`) demo. This topic describes this demo, which models the theory behind signal transmission:

1 Open the LPC Analysis and Synthesis of Speech demo by typing `dsplpcat` the MATLAB command line.
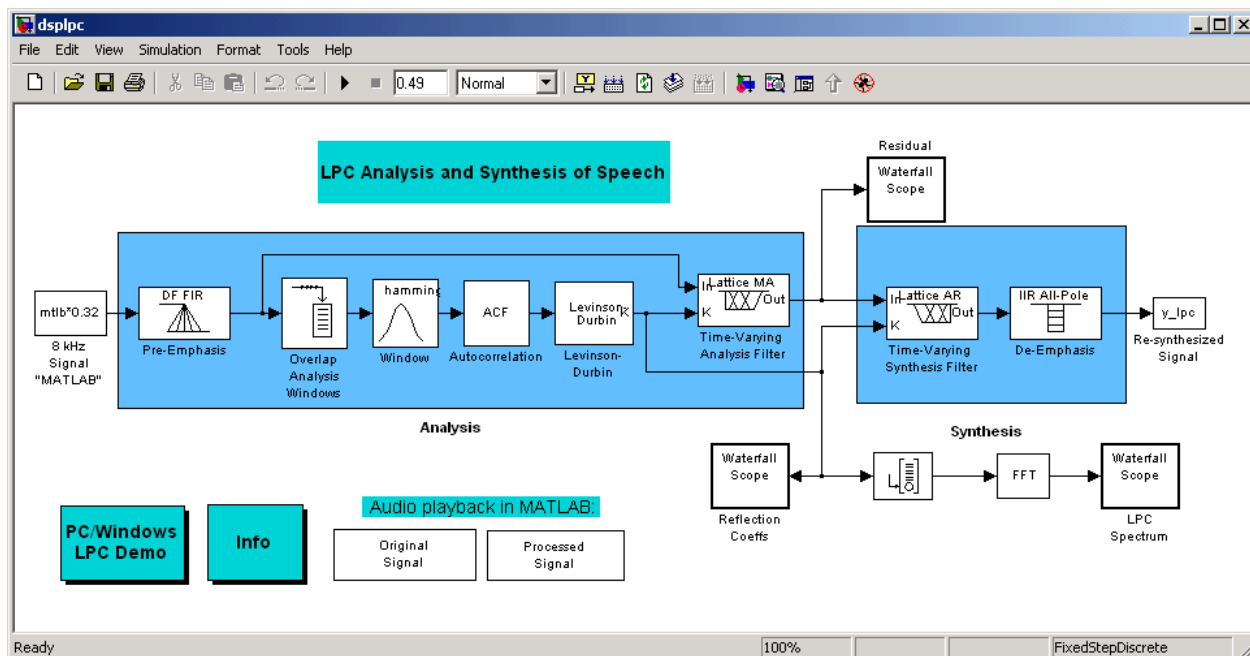
This model preemphasizes the input speech signal by applying an FIR filter. Then, it calculates the reflection coefficients of each frame using the Levinson-Durbin algorithm. The model uses these reflection coefficients to create the linear prediction analysis filter (lattice-structure). Next, the model 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 difference between the two because no quantization was performed. The model fully recovered the original signal.

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 (dsplpc) demo. 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 LPC Analysis and Synthesis of Speech demo by typing dsplpc at the MATLAB command line.

2 Save the dsplpc model file as scalar_quantizer_example.mdl in your working directory.

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.
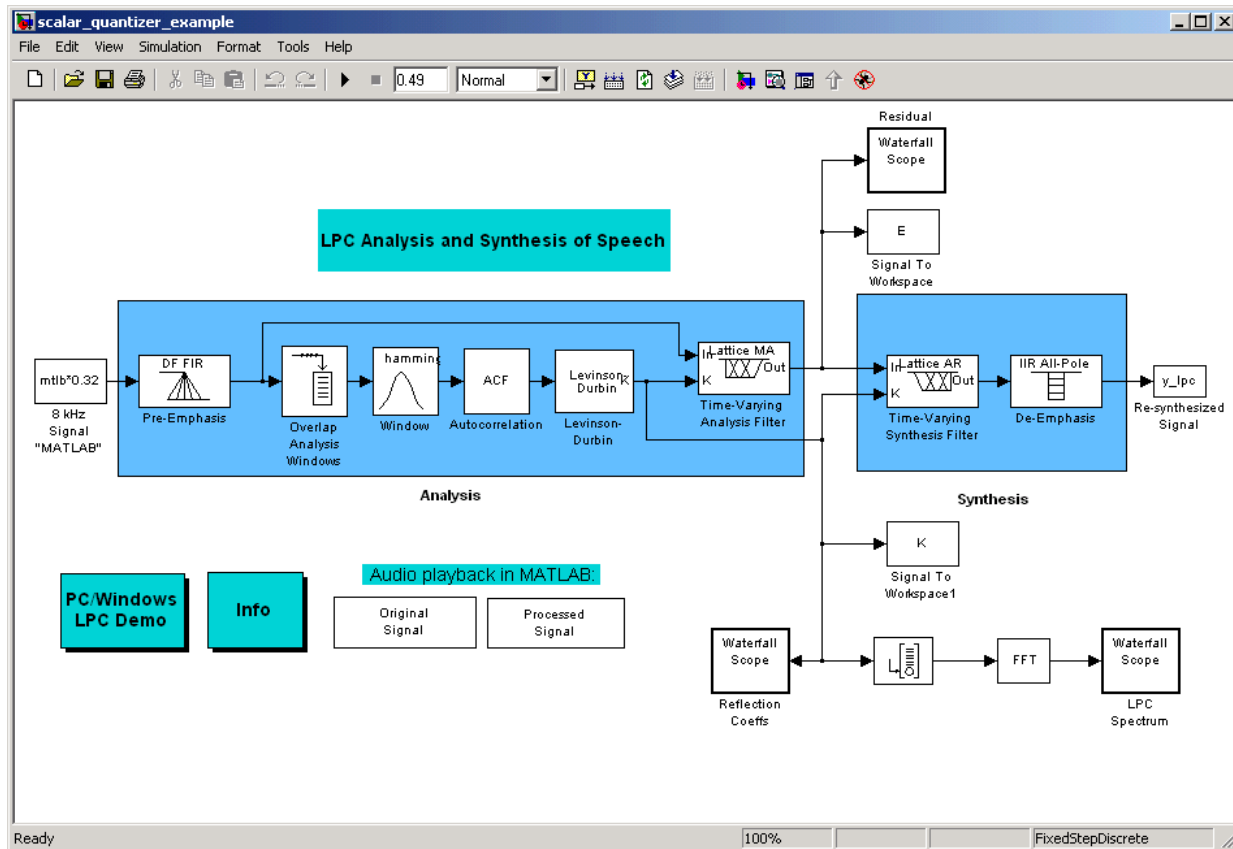
8 Run your model.

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_scalar_quantizer_example

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, scalar_quantizer_example.mdl, 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 Connect the blocks so your model looks similar to the following figure.
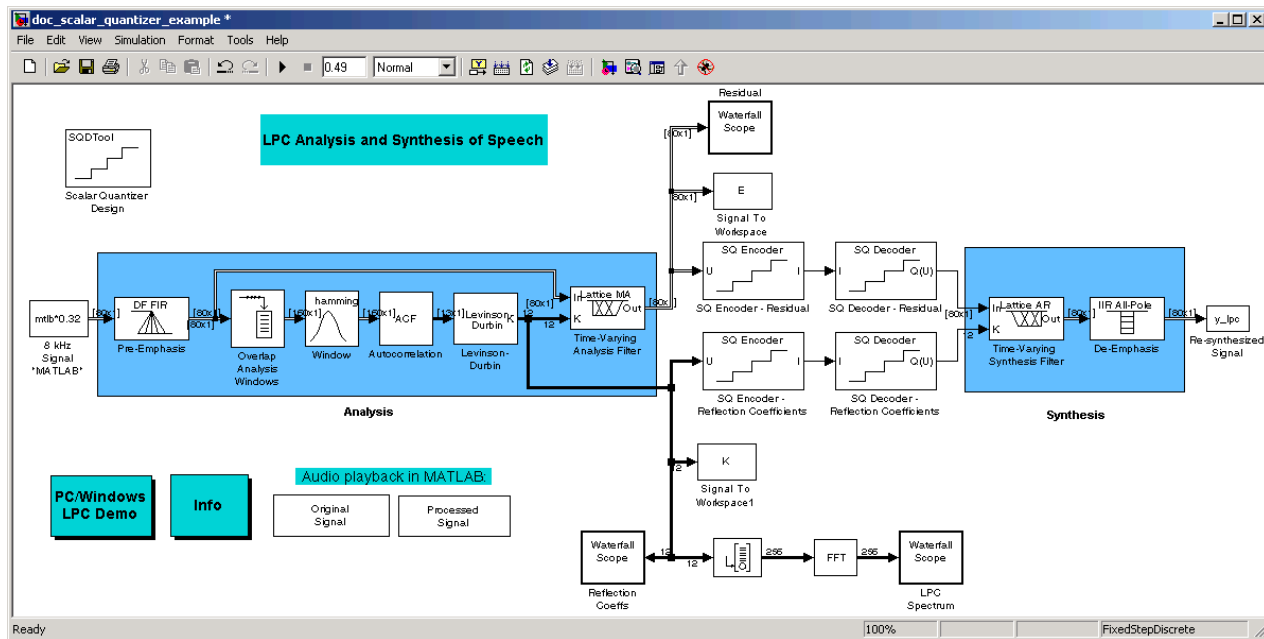
16 Run your model.

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 in the LPC Analysis and Synthesis of Speech demo model. 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 5-11.

Vector Quantizers

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 section, you quantize your reflection coefficients using vector quantizers to reduce the bit rate of your system.

This section includes the following topics:

- "Building Your Vector Quantizer Model" on page 5-11 — Reconfigure your scalar quantization model to use vector quantizers to quantize your reflection coefficients

- "Configuring and Running Your Model" on page 5-13 — Set your model parameters and use a split vector quantizer to quantize your reflection coefficients

Building Your Vector Quantizer Model

In this topic, you modify your scalar quantization model so that you are using a split vector quantizer to quantize your reflection coefficients:

1 If the model you created in "Creating a Scalar Quantizer" on page 5-5 is not open on your desktop, you can open an equivalent model by typing

    doc_scalar_quantizer_example2

at the MATLAB command prompt.

2 Delete the SQ Encoder - Reflection Coefficients and SQ Decoder - Reflection Coefficients blocks.

3 At the MATLAB command prompt, type dspcelpcoder.

The Signal Processing Blockset CELP-Based Vocoder demo opens. This demo quantizes linear prediction parameters using the split vector quantization method.

4 Double-click the CELP Encoder subsystem, and then double-click the
Frame Analysis subsystem. Copy the LSF Vector Quantization subsystem
and paste it in your model.

You use this subsystem to encode and decode your reflection coefficients
using the split vector quantization method.

5 From the Simulink library, and then from the Sinks library, click-and-drag
a Terminator block into your model.

6 From the Signal Processing Blockset library, from the Estimation library,
and then from the Linear Prediction library, click-and-drag a LSF/LSP to
LPC Conversion block and two LPC to/from RC blocks into your model.

7 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.
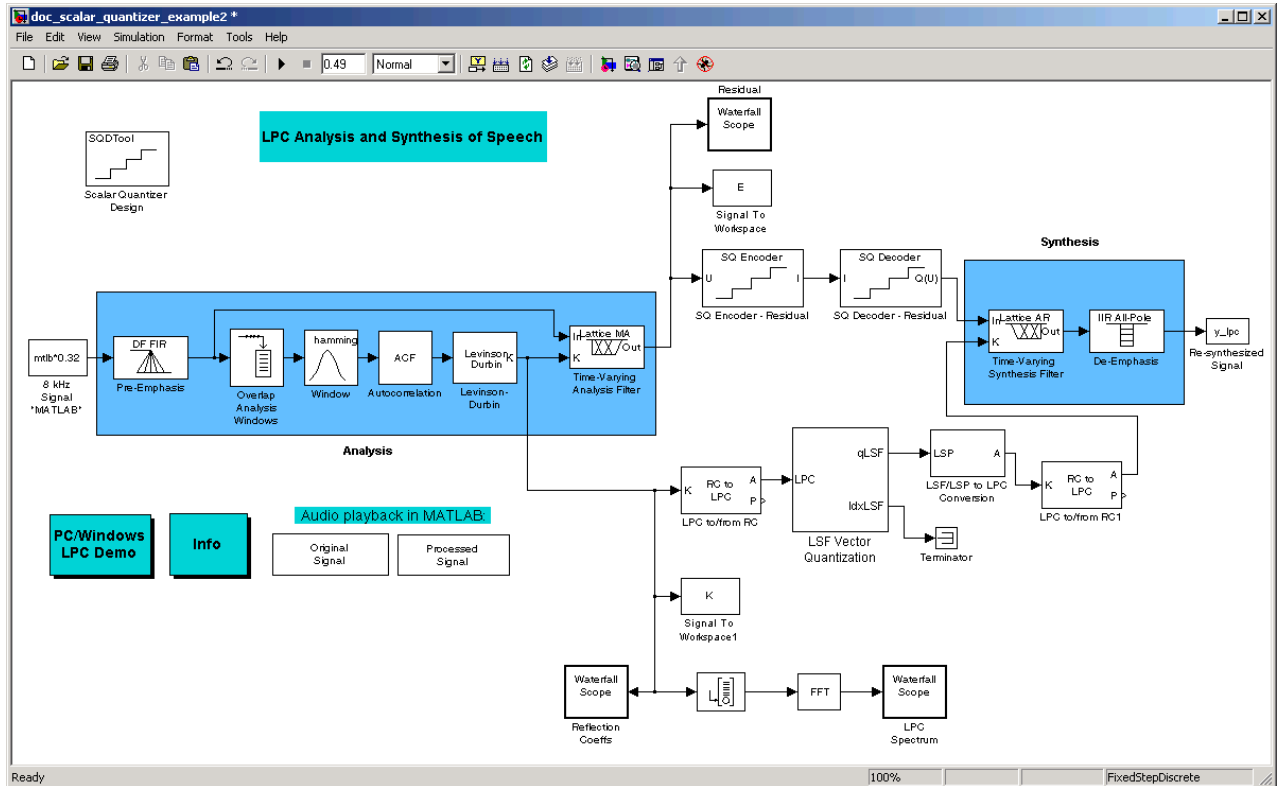
You have modified your model to include a subsystem capable of vector quantization. In the next topic, you reset your model parameters to quantize your reflection coefficients using the split vector quantization method.

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 5-11 is not open on your desktop, you can open an equivalent model by typing

```
doc_vector_quantizer_example
```

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.
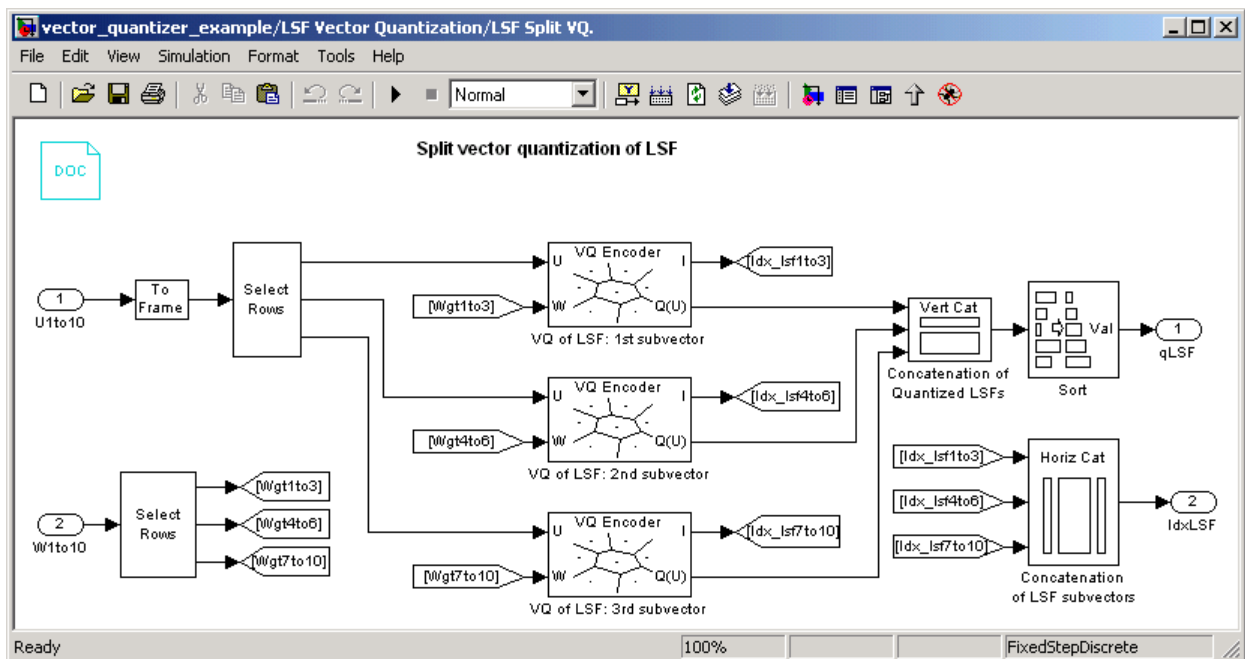
This subsystem divides each vector of 10 line spectral frequencies (LSFs), which represent your reflection coefficients, into three LSF subvectors. Each of these subvectors is sent to a separate vector quantizer. This method is called split vector quantization.

3 Double-click the VQ of LSF: 1st subvector block.

The **Block Parameters: VQ of LSF: 1st subvector** dialog box opens.
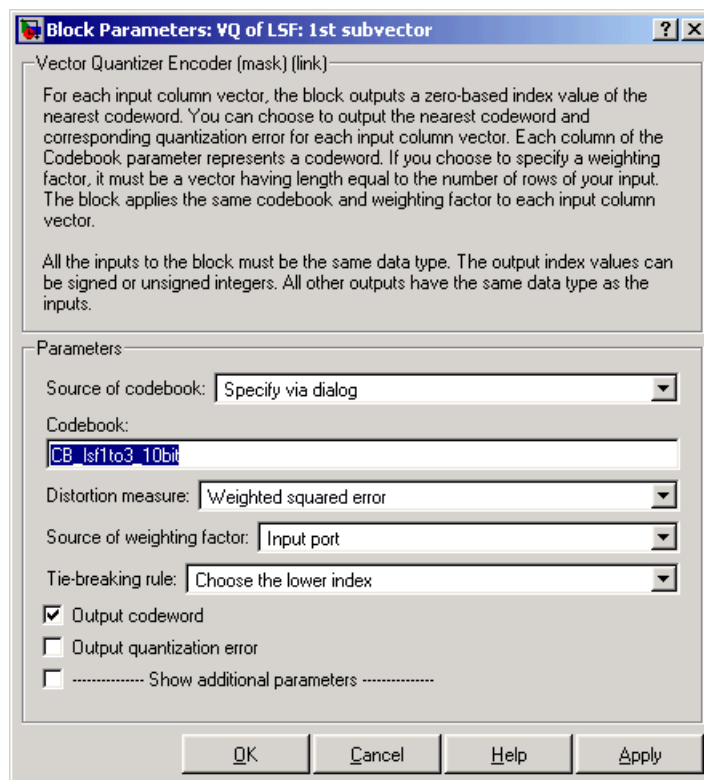
The variable `CB_lsf1to3_10bit` is the codebook for the subvector that contains the first three elements of the LSF vector. It is a 3-by-1024 matrix, where 3 is the number of elements in each codeword and 1024 is the number of codewords in the codebook. Because $2^{10} = 1024$, it takes 10 bits to quantize 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.
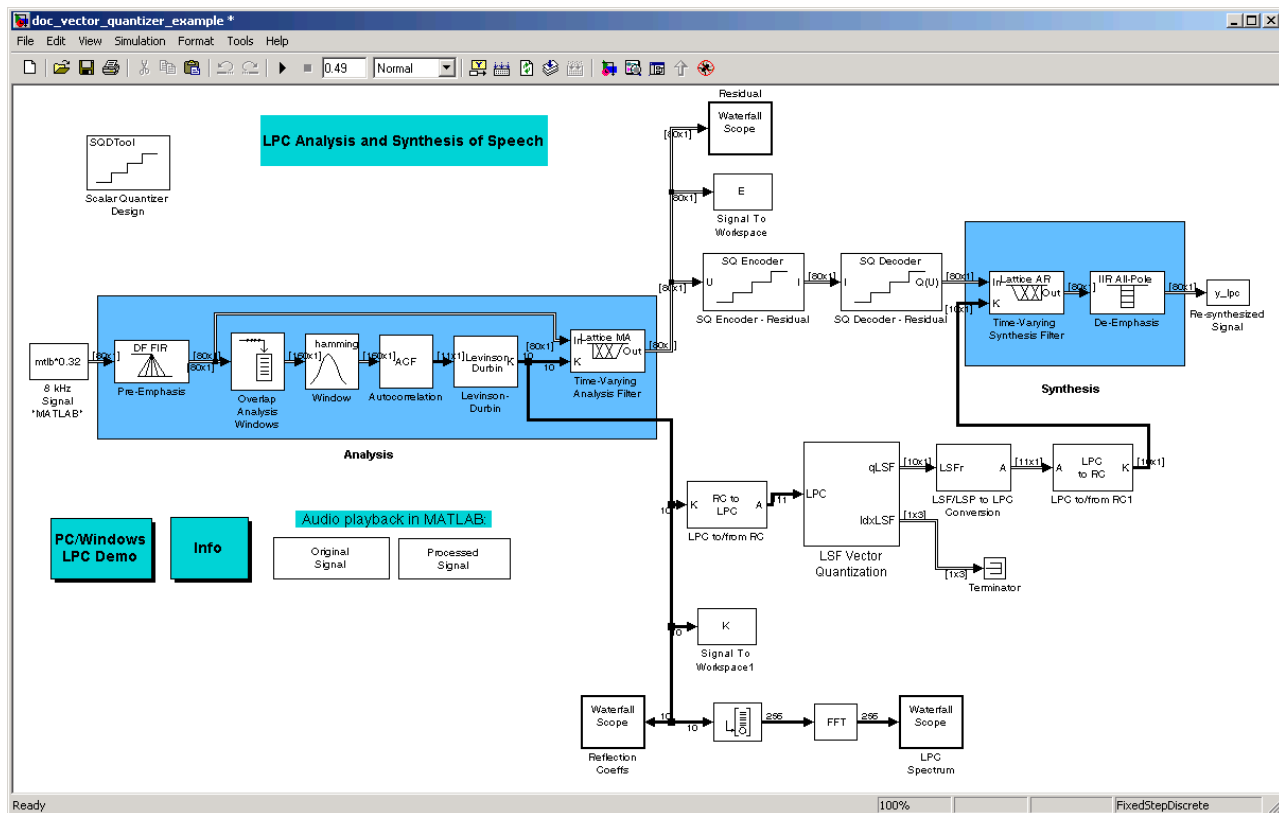
10 Double-click the Original Signal and Processed Signal blocks to listen to both the original and the processed signal.

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 LSF Vector Quantization subsystem to quantize the residual signal.

This example illustrates how you can use vector quantization to reduce the bit rate of your coder.

Statistics, Estimation, and Linear Algebra

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

Statistics

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 operations
- Running operations

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.

The following sections explain how basic mode and running mode differ:

- "Basic Operations" on page 6-3
- "Running Operations" on page 6-4

The statsdem demo illustrates the operation of several blocks from the Statistics library.

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.
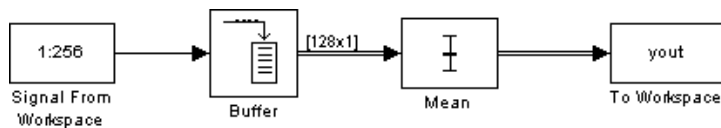
The Buffer block's **Buffer size** (M_o) parameter determines the size of the window. The **Buffer overlap** (L) parameter defines the "slide factor" for the window. At each sample instant, the window slides by M_o-L points. The **Buffer overlap** is often M_o-1 (the same as the Delay Line block), so 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.
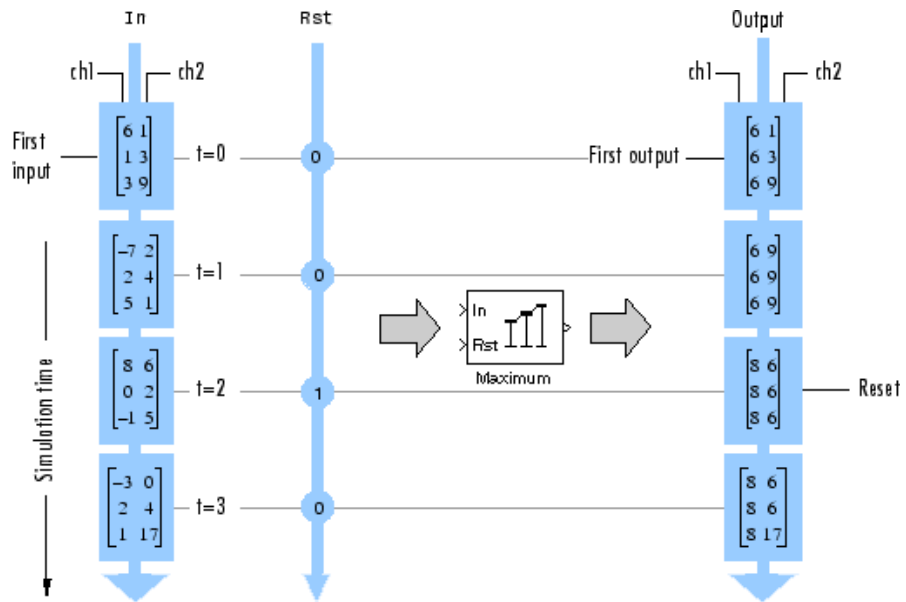
Power Spectrum Estimation

The Power Spectrum Estimation library provides a number of blocks for spectral analysis. Many of them have correlates in the Signal Processing Toolbox, 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.

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

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

Linear Algebra

The Matrices and Linear Algebra library provides three large sublibraries containing blocks for linear algebra:

- Linear System Solvers
- Matrix Factorizations
- Matrix Inverses

A third library, Matrix Operations, provides other essential blocks for working with matrices. See Chapter 1, "Working with Signals" for more information about matrix signals.

The following sections provide examples to help you get started with the linear algebra blocks:

- "Solving Linear Systems" on page 6-7
- "Factoring Matrices" on page 6-9
- "Inverting Matrices" on page 6-10

Solving Linear Systems

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

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

and finds x to be the vector [-2 0 1]'.
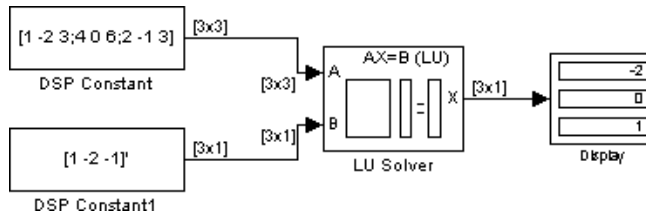
To build the model, set the following parameters:

- In the DSP Constant block, set **Constant value** = [1 -2 3;4 0 6;2 -1 3].

- In the DSP Constant1 block, set **Constant value** = [1 -2 -1]'.

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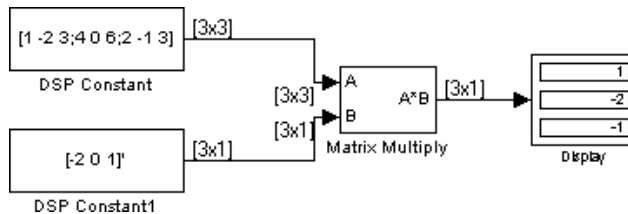
Factoring Matrices

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

$$A = \begin{bmatrix} 1 & -2 & 3 \\ 4 & 0 & 6 \\ 2 & -1 & 3 \end{bmatrix}$$
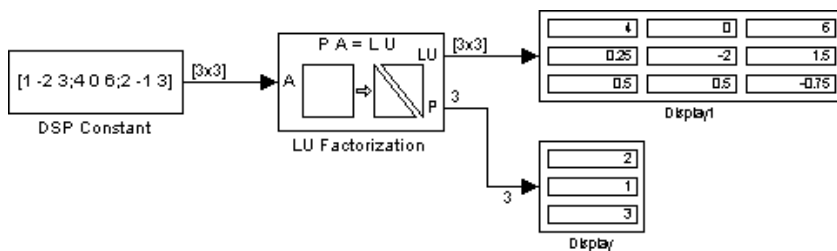
To build the model, in the DSP Constant block, set the **Constant value** parameter to [1 -2 3;4 0 6;2 -1 3].

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.

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

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.

$$U = \begin{bmatrix} 4 & 0 & 6 \\ 0 & -2 & 1.5 \\ 0 & 0 & -0.75 \end{bmatrix} \qquad L = \begin{bmatrix} 1 & 0 & 0 \\ 0.25 & 1 & 0 \\ 0.5 & 0.5 & 1 \end{bmatrix}$$

You can check that $LU = A_p$ with the Matrix Multiply block, as shown in the model below.
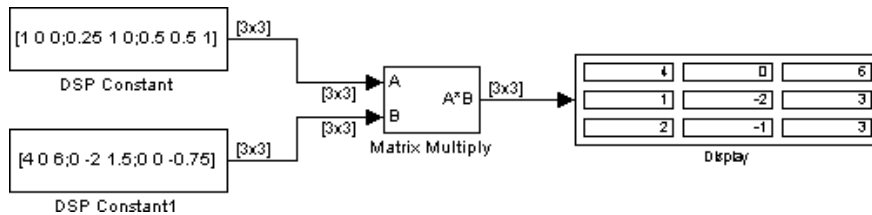
Inverting Matrices

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

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

and then forms the product $A^{-1}A$, which yields the identity matrix of order 3, as expected.
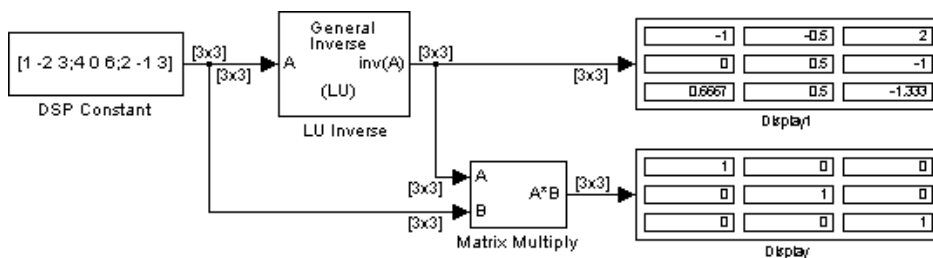
To build the model, in the DSP Constant block, set the **Constant value** parameter to [1 -2 3;4 0 6;2 -1 3].

As shown above, the computed inverse is

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

7

Data Type Support

All Signal Processing Blockset blocks support the single- and double-precision floating-point data type. Many blocks support other data types.

Supported Data Types and How to Convert to Them

> **Note** All data type support applies to both simulation and Real-Time
> Workshop C code generation. All Signal Processing Blockset blocks support
> single- and double-precision floating point.

The following table lists all data types supported by the Signal Processing
Blockset, and how to convert to these data types. To see which data types
a particular block supports, see the "Supported Data Types" section in the
block's reference page.

Supported Data Types and How to Convert to Them

Data Types Supported by Signal Processing Blockset Blocks	Commands and Blocks for Converting Data Types	Comments
Double-precision floating point	• `double` • Data Type Conversion block	Simulink built-in data type supported by all Signal Processing Blockset blocks.
Single-precision floating point	• `single` • Data Type Conversion block	Simulink built-in data type supported by all Signal Processing Blockset blocks.
Boolean	• Data Type Conversion block	Simulink built-in data type. To learn more, see "Boolean Support" on page 7-15.
Integer (8-,16-, or 32-bits)	• `int8`, `int16`, `int32` • Data Type Conversion block	Simulink built-in data type
Unsigned integer (8-,16-, or 32-bits)	• `uint8`, `uint16`, `uint32` • Data Type Conversion block	Simulink built-in data type

Supported Data Types and How to Convert to Them (Continued)

Data Types Supported by Signal Processing Blockset Blocks	Commands and Blocks for Converting Data Types	Comments
Fixed-point data types	• Data Type Conversion block • Simulink Fixed Point `num2fixpt` function • Functions and GUIs for designing quantized filters with the Filter Design Toolbox (compatible with Filter Realization Wizard block)	To learn more about fixed-point data types in the Signal Processing Blockset, see Chapter 8, "Working with Fixed-Point Data".
Custom data types	See "Correctly Defining Custom Data Types" on page 7-14 to learn about custom data types.	

Block Data Type Support Table

The following table shows what data types are accepted on the main input data ports of each Signal Processing Blockset block. If the block is a source, the table shows what data types are accepted on the main output data ports of each source block.

If the **Double**, **Single**, **Boolean**, and/or **Custom Data Types** columns are populated by a x, the block supports those data types.

- If the **Base Integer** and/or **Fixed-Point** columns are populated with an s, the block supports signed integers and/or fixed-point data types.

- If the **Base Integer** and/or **Fixed-Point** columns are populated with a u, the block supports unsigned integers and/or fixed-point data types.

All blocks in the Signal Processing Blockset support code generation.

Block	Double	Single	Boolean	Base Integer	Fixed-Point	Custom Data Types
Analog Filter Design	x					
Analytic Signal	x	x				
Autocorrelation	x	x		s	s	
Autocorrelation LPC	x	x				
Backward Substitution	x	x				
Block LMS Filter	x	x				
Buffer	x	x	x	s,u	s	x
Burg AR Estimator	x	x				
Burg Method	x	x				
Check Signal Attributes	x	x	x	s,u	s	x
Chirp	x	x				

Block	Double	Single	Boolean	Base Integer	Fixed-Point	Custom Data Types
Cholesky Factorization	x	x				
Cholesky Inverse	x	x				
Cholesky Solver	x	x				
CIC Decimation				s	s	
CIC Interpolation				s	s	
Complex Cepstrum	x	x				
Complex Exponential	x	x				
Constant Diagonal Matrix	x	x		s,u	s,u	
Constant Ramp	x	x		s,u	s,u	
Convert 1-D to 2-D	x	x	x	s,u	s,u	x
Convert 2-D to 1-D	x	x	x	s,u	s,u	x
Convolution	x	x		s	s	
Correlation	x	x		s	s	
Counter	x	x	x	s,u		
Covariance AR Estimator	x	x				
Covariance Method	x	x				
Create Diagonal Matrix	x	x	x	s,u	s,u	x
Cumulative Product	x	x		s	s	
Cumulative Sum	x	x		s	s	
Data Type Conversion	Simulink block					
dB Conversion	x	x				
dB Gain	x	x		s,u	s,u	

Block	Double	Single	Boolean	Base Integer	Fixed-Point	Custom Data Types
DCT	x	x				
Delay	x	x	x	s,u	s,u	x
Delay Line	x	x	x	s,u	s,u	x
Detrend	x	x				
Difference	x	x		s	s	
Digital Filter	x	x		s	s	
Digital Filter Design	x	x				
Discrete Impulse	x	x	x	s,u	s,u	
Display	Simulink block					
Downsample	x	x	x	s,u	s,u	x
DSP Constant	x	x	x	s,u	s,u	x
DWT	x	x				
Dyadic Analysis Filter Bank	x	x				
Dyadic Synthesis Filter Bank	x	x				
Edge Detector	x	x	x	s,u	s,u	x
Event-Count Comparator	x	x	x	s,u	s,u	x
Extract Diagonal	x	x	x	s,u	s,u	x
Extract Triangular Matrix	x	x	x	s,u	s,u	x
Fast Block LMS Filter	x	x				
FFT	x	x		s	s	
Filter Realization Wizard	x	x		s,u	s,u	
FIR Decimation	x	x		s	s	

Block	Double	Single	Boolean	Base Integer	Fixed-Point	Custom Data Types
FIR Interpolation	x	x		s	s	
FIR Rate Conversion	x	x		s	s	
Flip	x	x	x	s,u	s,u	x
Forward Substitution	x	x				
Frame Conversion	x	x	x	s,u	s	x
From Wave Device	x	x		s 16-bit u 8-bit		
From Wave File	x	x		s 16-bit u 8-bit		
G711 Codec				s 16-bit		
Histogram	x	x		s,u	s,u	
IDCT	x	x				
Identity Matrix	x	x	x	s,u	s,u	x
IDWT	x	x				
IFFT	x	x		s	s	
Inherit Complexity	x	x	x	s,u	s,u	x
Interpolation	x	x				
Inverse Short-Time FFT	x	x				
Kalman Adaptive Filter	x	x				
LDL Factorization	x	x				
LDL Inverse	x	x				
LDL Solver	x	x				
Least Squares Polynomial Fit	x	x				
Levinson-Durbin	x	x		s	s	

Block	Double	Single	Boolean	Base Integer	Fixed-Point	Custom Data Types
LMS Adaptive Filter	x	x				
LMS Filter	x	x				
LPC to LSF/LSP Conversion	x	x				
LSF/LSP to LPC Conversion	x	x				
LPC to/from Cepstral Coefficients	x	x				
LPC to/from RC	x	x				
LPC/RC to Autocorrelation	x	x				
LU Factorization	x	x				
LU Inverse	x	x				
LU Solver	x	x				
Magnitude FFT	x	x		s	s	
Matrix 1-Norm	x	x		s	s	
Matrix Concatenation	Simulink block					
Matrix Exponential	x	x				
Matrix Multiply	x	x	x	s,u	s,u	
Matrix Product	x	x		s,u	s,u	
Matrix Scaling	x	x		s	s	
Matrix Square	x	x				
Matrix Sum	x	x		s,u	s,u	
Matrix Viewer	x	x	x	s,u	s,u	x
Maximum	x	x		s,u	s,u	
Mean	x	x		s	s	

Block	Double	Single	Boolean	Base Integer	Fixed-Point	Custom Data Types
Median	x	x		s,u	s,u	
Minimum	x	x		s,u	s,u	
Modified Covariance AR Estimator	x	x				
Modified Covariance Method	x	x				
Multiphase Clock	x	x	x			
Multiport Selector	x	x	x	s,u	s,u	x
N-Sample Enable	x		x			
N-Sample Switch	x	x	x	s,u	s,u	x
Normalization	x	x		s	s	
Offset	x	x		s	s	
Overlap-Add FFT Filter	x	x				
Overlap-Save FFT Filter	x	x				
Overwrite Values	x	x	x	s,u	s,u	x
Pad	x	x	x	s,u	s,u	x
Peak Finder	x	x		s,u	s,u	
Periodogram	x	x				
Permute Matrix	x	x	x	s,u	s,u	x
Polynomial Evaluation	x	x				
Polynomial Stability Test	x	x				
Pseudoinverse	x	x				
QR Factorization	x	x				

Block	Double	Single	Boolean	Base Integer	Fixed-Point	Custom Data Types
QR Solver	x	x				
Quantizer	Simulink block					
Queue	x	x	x	s,u	s,u	x
Random Source	x	x				
Real Cepstrum	x	x				
Reciprocal Condition	x	x				
Repeat	x	x	x	s,u	s,u	x
RLS Adaptive Filter	x	x				
RLS Filter	x	x				
RMS	x	x				
Sample and Hold	x	x	x	s,u	s,u	x
Scalar Quantizer Decoder				s,u	s,u	
Scalar Quantizer Design	x					
Scalar Quantizer Encoder	x	x		s	s	
Selector	Simulink block					
Short-Time FFT	x	x		s	s	
Signal From Workspace	x	x		s,u	s,u	
Signal To Workspace	x	x	x	s,u	s,u	x
Sine Wave	x	x		s	s	
Singular Value Decomposition	x	x				
Sort	x	x		s,u	s,u	
Spectrum Scope	x	x	x	s,u	s,u	x

Block	Double	Single	Boolean	Base Integer	Fixed-Point	Custom Data Types
Stack	x	x	x	s,u	s,u	x
Standard Deviation	x	x				
Submatrix	x	x	x	s,u	s,u	x
SVD Solver	x	x				
Time Scope	Simulink block					
Toeplitz	x	x	x	s,u	s,u	x
To Wave Device	x	x		s 16-bit u 8-bit		
To Wave File	x	x		s 16-bit u 8-bit		
Transpose	x	x	x	s,u	s,u	x
Triggered Delay Line	x	x	x	s,u	s,u	x
Triggered Signal From Workspace	x	x		s,u	s,u	
Triggered To Workspace	x	x	x	s,u	s,u	x
Two-Channel Analysis Subband Filter	x	x		s	s	
Two-Channel Synthesis Subband Filter	x	x		s	s	
Unbuffer	x	x	x	s,u	s,u	x
Uniform Decoder				s,u		
Uniform Encoder	x	x				
Unwrap	x	x				
Upsample	x	x	x	s,u	s,u	x

Block	Double	Single	Boolean	Base Integer	Fixed-Point	Custom Data Types
Variable Fractional Delay	x	x				
Variable Integer Delay	x	x	x	s,u	s,u	x
Variable Selector	x	x	x	s,u	s,u	x
Variance	x	x		s	s	
Vector Quantizer Decoder				s,u	s,u	
Vector Quantizer Design	x					
Vector Quantizer Encoder	x	x		s	s	
Vector Scope	x	x	x	s,u	s,u	x
Waterfall	x	x		s,u	s,u	
Window Function	x	x		s	s	
Yule-Walker AR Estimator	x	x				
Yule-Walker Method	x	x				
Zero Crossing	x	x		s,u	s,u	
Zero Pad	x	x	x	s,u	s	x

Viewing Data Types of Signals In Models

You can enable data type labels of the signals in your model. In the model window, from the **Format** menu, point to **Port/Signal Displays**, and select **Port Data Types**. Now, the signal lines in the model have labels indicating their data types. To see the labels, you may have to refresh the model diagram. To do this, from the **Edit** menu, select **Update Diagram**.
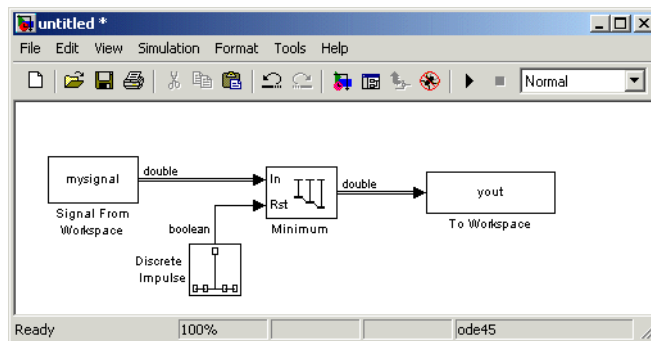
Signal Lines Labeled with Their Data Types

Correctly Defining Custom Data Types

Custom data types are user-defined data types. You must define your custom data types by following the guidelines provided in the topic on custom data types in the Writing S-Functions Simulink documentation. If you do not follow the Simulink guidelines for creating custom data types, the Signal Processing Blockset blocks may not properly support your custom data types.

Boolean Support

Many Signal Processing Blockset blocks accept or output logical signals. All such blocks support the Boolean data type at their appropriate ports:

- All block input ports that accept logical signals support the Boolean data type.

- The default data type of all outputs that are logical signals is Boolean. You can change this default setting and disable Boolean support as described in "Effects of Enabling and Disabling Boolean Support" on page 7-17.

The following topics provide more information on Boolean data type support:

- "Advantages of Using the Boolean Data Type" on page 7-15
- "Lists of Blocks Supporting Boolean Inputs or Outputs" on page 7-15
- "Effects of Enabling and Disabling Boolean Support" on page 7-17
- "Steps to Disabling Boolean Support" on page 7-18

Advantages of Using the Boolean Data Type

Using the Boolean data type rather than floating-point data types speeds up simulations and results in smaller, faster generated C code. For more on generated code, see "Code Generation" in the Getting Started with Signal Processing Blockset documentation.

Lists of Blocks Supporting Boolean Inputs or Outputs

The following blocks have reset ports that accept the Boolean data type:

Counter	Minimum
Cumulative Product	N-Sample Enable
Cumulative Sum	N-Sample Switch
Delay	RMS
Histogram	Standard Deviation

Maximum	Variance
Mean	

The following blocks have input ports that accept the Boolean data type:

Buffer	Repeat
Check Signal Attributes	Sample and Hold
Convert 1-D to 2-D	Signal To Workspace
Convert 2-D to 1-D	Spectrum Scope
Create Diagonal Matrix	Stack
Delay Line	Submatrix
Downsample	Time Scope
Extract Triangular Matrix	Toeplitz
Flip	Transpose
Frame Conversion	Triggered Delay Line
Identity Matrix	Triggered To Workspace
Inherit Complexity	Unbuffer
Matrix Viewer	Upsample
Multiport Selector	Variable Integer Delay
Overwrite Values	Variable Selector
Pad	Vector Scope
Permute Matrix	Zero Pad
Queue	

Some or all of the output ports of the following blocks support outputs with the Boolean data type:

Buffer	Multiport Selector
Check Signal Attributes	N-Sample Enable

Convert 1-D to 2-D	Overwrite Values
Convert 2-D to 1-D	Pad
Counter	Permute Matrix
Create Diagonal Matrix	Polynomial Stability Test
Delay Line	Queue
Downsample	Repeat
Edge Detector	Sample and Hold
Event-Count Comparator	Scalar Quantizer Encoder
Extract Diagonal	Stack
Extract Triangular Matrix	Submatrix
Flip	Toeplitz
Frame Conversion	Transpose
From Wave File	Triggered Delay Line
Identity Matrix	Unbuffer
Inherit Complexity	Upsample
LPC to/from RC	Variable Integer Delay
LPC to LSF/LSP Conversion	Variable Selector
LU Factorization	Zero Pad
Multiphase Clock	

Effects of Enabling and Disabling Boolean Support

By default, Simulink *enables* Boolean support. When you leave Boolean support enabled, all Boolean-supporting output ports *always* output the Boolean data type.

In some cases, you may want to override the Simulink default and *disable* Boolean support. For example, you may have a model that you created *before* Boolean support existed. Leaving the Boolean support enabled in this model may cause some blocks that used to output the double-precision data type to

output the Boolean data type. If the introduction of the Boolean data type breaks your model, you can fix the problem by disabling Boolean support.

The following table describes the effects of enabling and disabling Boolean support. Note that when you *disable* Boolean support, some Boolean-supporting output ports output double-precision data.

Type of Boolean-Supporting Output Port	Effect of Enabling Boolean Support (Default)	Effect of Disabling Boolean Support
• On a block with at least one input port • Did not support the Boolean data type in versions of the Signal Processing Blockset before Version 5.0 (For example, the Edge Detector block)	Output is *always* Boolean, regardless of the input data type.	• When input is double precision, the output is also double precision. • When input is *not* double precision, the output is Boolean.
With a corresponding block parameter for setting output data type to `Logical` or `Boolean` (for example, in the N-Sample Enable block)	Output is *always* Boolean, regardless of whether you set the output port to `Logical` or `Boolean`.	• When set to `Logical`, the output is double precision. • When set to `Boolean`, the output is Boolean.

Steps to Disabling Boolean Support

To disable Boolean data type support in a particular model, clear the Boolean-enabling configuration parameter in the model by completing the following:

- "Step 1: Open the Configuration Parameters Dialog Box" on page 7-19
- "Step 2: Disable the Boolean Data Type in the Advanced Tab" on page 7-19
- "Step 3: (Optional) Verify Data Types of Signals" on page 7-20

You can also set Simulink simulation preferences so that *all* new models you create have Boolean support disabled. For more information, see "Setting Simulink Preferences" in the Simulink Getting Started documentation.

Step 1: Open the Configuration Parameters Dialog Box

In the model for which you want to enable Boolean data type support, from the **Simulation** menu, select **Configuration Parameters**. The Configuration Parameters dialog box opens.

The following figure illustrates the Configuration Parameters dialog box with the appropriate settings for signal processing simulations (note the discrete Fixed-step solver setting).
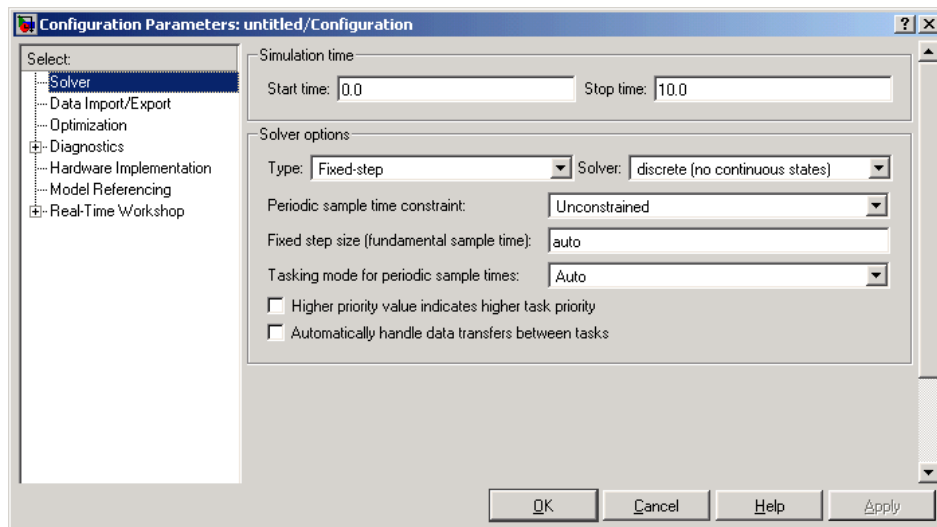
Step 2: Disable the Boolean Data Type in the Advanced Tab

Open the Configuration Parameters dialog box. In the **Select** pane, click **Optimization**. Clear the **Implement logic signals as boolean data (vs. double)** check box. Click **OK**.

You have now disabled Boolean support in your model; for certain cases, output ports that support the Boolean data type will output double-precision

data rather than Boolean data, as explained in "Effects of Enabling and Disabling Boolean Support" on page 7-17.
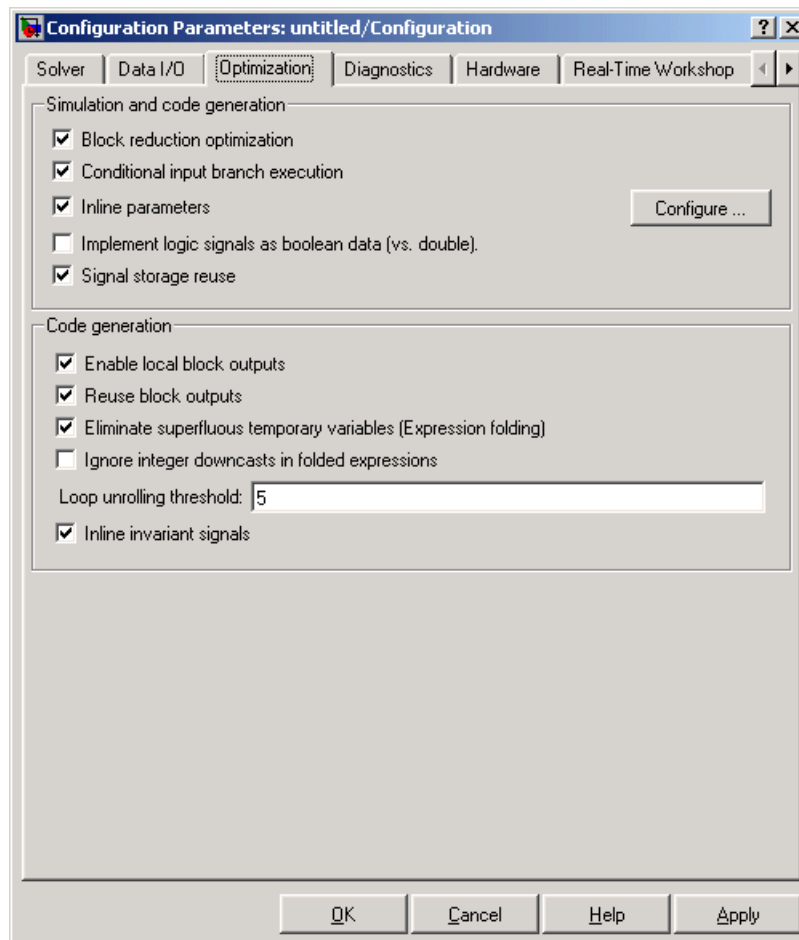
Step 3: (Optional) Verify Data Types of Signals

Check the data types of the signals in the model by turning on the automatic labeling of signal data types (see "Viewing Data Types of Signals In Models" on page 7-13). Some Boolean-supporting output ports might have output signals labeled double rather than boolean, depending on whether the inputs

to the block are double-precision (see "Effects of Enabling and Disabling Boolean Support" on page 7-17).

If you do not see the data type labels after turning them on, you may have to refresh the model diagram by selecting the **Edit** menu in your model and then selecting **Update diagram**.

8

Working with Fixed-Point Data

Fixed-Point Signal Processing Development

Many of the blocks in the Signal Processing Blockset have fixed-point support, so you can design signal processing systems that use fixed-point arithmetic. Fixed-point support in the Signal Processing Blockset 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. The generated code uses all allowed data types supported by the embedded target, and automatically includes all necessary shift and scaling operations

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

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 the Signal Processing Blockset

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

The Signal Processing Blockset 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 the Signal Processing Blockset and Real-Time Workshop produces code ready for execution on a fixed-point processor. All the choices you make in simulation with the Signal Processing Blockset 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 documentation.

Fixed-Point Signal Processing Applications

Fixed-point support in the Signal Processing Blockset 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

Blocks with Fixed-Point Support

The following table lists all of the blocks in the Signal Processing Blockset that support fixed-point data types in some or all modes. These blocks are colored orange in the Signal Processing Blockset library. To take full advantage of the fixed-point capabilities of the following blocks, you must install Simulink Fixed Point.

Signal Processing Blockset Blocks with Fixed-Point Support

Autocorrelation	Buffer	Check Signal Attributes	CIC Decimation
CIC Interpolation	Constant Diagonal Matrix	Constant Ramp	Convert 1-D to 2-D
Convert 2-D to 1-D	Convolution	Correlation	Counter
Create Diagonal Matrix	Cumulative Product	Cumulative Sum	Data Type Conversion (Simulink block)
dB Gain	DCT	Delay	Delay Line
Difference	Digital Filter	Discrete Impulse	Display (Simulink block)
Downsample	DSP Constant	Edge Detector	Event-Count Comparator
Extract Diagonal	Extract Triangular Matrix	FFT	Filter Realization Wizard
FIR Decimation	FIR Interpolation	FIR Rate Conversion	Flip
Frame Conversion	G711 Codec	Histogram	IDCT
Identity Matrix	IFFT	Inherit Complexity	Levinson-Durbin
LMS Filter	Magnitude FFT	Matrix-1 Norm	Matrix Concatenation (Simulink block)
Matrix Product	Matrix Scaling	Matrix Sum	Matrix Viewer
Maximum	Mean	Median	Minimum
Multiphase Clock	Multiport Selector	N-Sample Enable	N-Sample Switch
Normalization	Offset	Overwrite Values	Pad

Signal Processing Blockset Blocks with Fixed-Point Support (Continued)

Peak Finder	Permute Matrix	Queue	Repeat
Sample and Hold	Scalar Quantizer Decoder	Scalar Quantizer Encoder	Selector (Simulink block)
Short-Time FFT	Signal From Workspace	Signal To Workspace	Sine Wave
Sort	Spectrum Scope	Stack	Submatrix
Time Scope (Simulink block)	Toeplitz	Transpose	Triggered Delay Line
Triggered Signal From Workspace	Triggered To Workspace	Two-Channel Analysis Subband Filter	Two-Channel Synthesis Subband Filter
Unbuffer	Upsample	Variable Integer Delay	Variable Selector
Variance	Vector Quantizer Decoder	Vector Quantizer Encoder	Vector Scope
Waterfall	Window Function	Zero Crossing	Zero Pad

Concepts and Terminology

This section gives an overview of fixed-point concepts and terminology that you might want to refer to as you take advantage of fixed-point support in the Signal Processing Blockset:

- "Fixed-Point Data Types" on page 8-8
- "Scaling" on page 8-9
- "Precision and Range" on page 8-10

The Glossary on page 1 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:
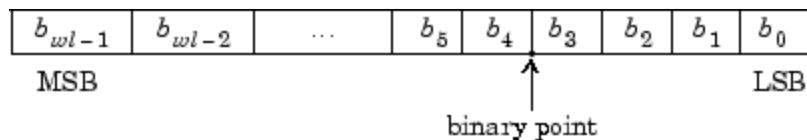
where

- 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 the Signal Processing Blockset. See "Two's Complement" on page 8-14 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 the Signal Processing Blockset, 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's [Slope Bias] representation that has a bias equal to zero and a slope adjustment equal to one. This is referred to as binary point-only scaling or power-of-two scaling:

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

or

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

In the Signal Processing Blockset, 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 Signal Processing Blockset supports binary point-only scaling, so the whole of the data type and scaling information is contained in these two quantities. This is in contrast to Simulink Fixed Point, which supports [Slope Bias] scaling in its full generality.

Precision and Range

You must pay attention to the precision and range of the fixed-point data types and scalings you choose for the blocks in your simulations, in order to know whether rounding methods will be invoked or if overflows will occur.

Range

The range is the span of numbers that a fixed-point data type and scaling can represent. The range of representable numbers for a two's complement fixed-point number of word length wl, scaling S, and bias B is illustrated below:
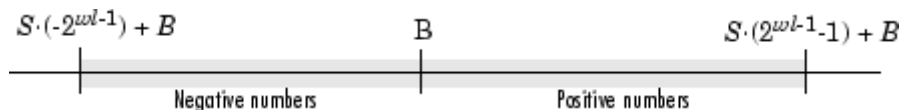
For both signed and unsigned fixed-point numbers of any data type, the number of different bit patterns is 2^{wl}.

For example, in two's complement, negative numbers must be represented as well as zero, so the maximum value is $2^{wl-1}-1$. Because there is only one representation for zero, there are an unequal number of positive and negative numbers. This means there is a representation for -2^{wl-1} but not for 2^{wl-1}:

For Slope = 1 and Bias = 0:
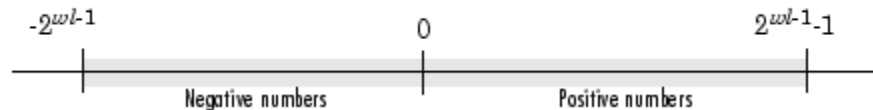
Overflow Handling. Because a fixed-point data type represents numbers within a finite range, overflows can occur if the result of an operation is larger or smaller than the numbers in that range.

The Signal Processing Blockset does not allow you to add guard bits to a data type on-the-fly in order to avoid overflows. Any guard bits must be allocated upon model initialization. However, the Signal Processing Blockset does allow you to either *saturate* or *wrap* overflows. Saturation represents positive overflows as the largest positive number in the range being used, and negative overflows as the largest negative number in the range being used. Wrapping uses modulo arithmetic to cast an overflow back into the representable range of the data type. See "Modulo Arithmetic" on page 8-13 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 Methods. One of the limitations of representing numbers with finite precision is that not every number in the available range can be represented exactly. When the result of a fixed-point calculation is a number that cannot be represented exactly by the data type and scaling being used, precision is lost. A rounding method must be used to cast the result to a representable number. The Signal Processing Blockset currently supports `Floor` and `Nearest` rounding methods.

`Floor`, which is equivalent to truncation, rounds the output of a calculation to the closest representable number in the direction of negative infinity.

`Nearest` rounds the output of a calculation to the closest representable number, with the exact midpoint rounded to the closest representable number in the direction of positive infinity.

Arithmetic Operations

The following sections describe the arithmetic operations used by fixed-point Signal Processing Blockset blocks:

- "Modulo Arithmetic" on page 8-13
- "Two's Complement" on page 8-14
- "Addition and Subtraction" on page 8-15
- "Multiplication" on page 8-16
- "Casts" on page 8-18

These sections will help you understand what data type and scaling choices result in overflows or a loss of precision.

Modulo Arithmetic

Binary math is based on modulo arithmetic. Modulo arithmetic uses only a finite set of numbers, wrapping the results of any calculations that fall outside the given set back into the set.

For example, the common everyday clock uses modulo 12 arithmetic. Numbers in this system can only be 1 through 12. Therefore, in the "clock" system, 9 plus 9 equals 6. This can be more easily visualized as a number circle:

9...   ...plus 9 more....

...equals 6.
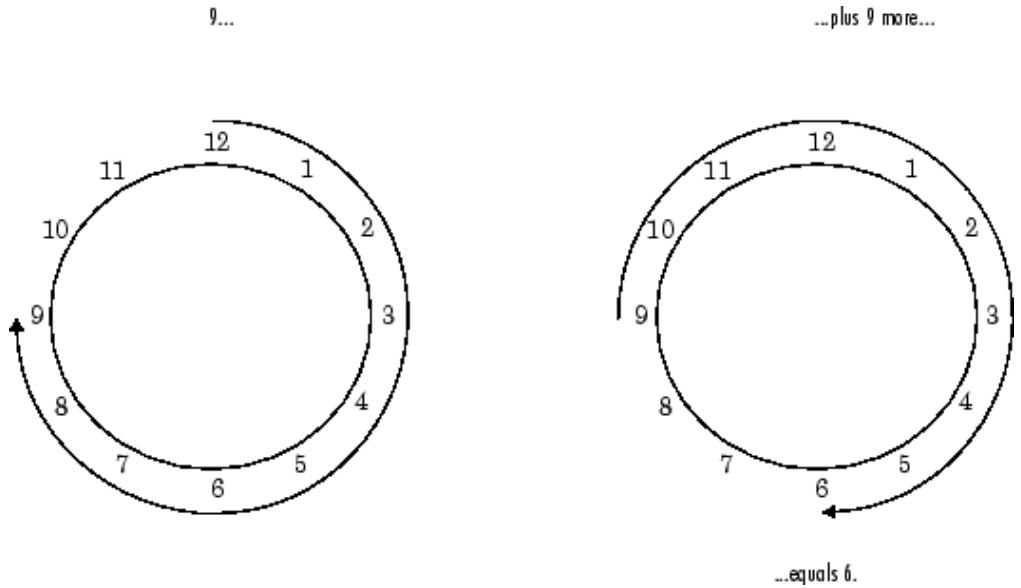
Similarly, binary math can only use the numbers 0 and 1, and any arithmetic results that fall outside this range are wrapped "around the circle" to either 0 or 1.

Two's Complement

Two's complement is a way to interpret a binary number. In two's complement, positive numbers always start with a 0 and negative numbers always start with a 1. If the leading bit of a two's complement number is 0, the value is obtained by calculating the standard binary value of the number. If the leading bit of a two's complement number is 1, the value is obtained by assuming that the leftmost bit is negative, and then calculating the binary value of the number. For example,

$$01 \quad = \quad (0 + 2^0) = 1$$
$$11 \quad = \quad ((-2^1) + (2^0)) = (-2 + 1) = -1$$

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:

$$11010 \longrightarrow 00101$$

Next, add a 1, wrapping all numbers to 0 or 1:

```
 00101
  +1
 00110 (6)
```

Addition and Subtraction

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

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

```
 010010.1    (18.5)
+ 0110.110   (6.75)
 011001.010  (25.25)
```

Fixed-point subtraction is equivalent to adding while using the two's complement value for any negative values. In subtraction, the addends must be sign extended to match each other's length. For example, consider subtracting 0110.110 (6.75) from 010010.1 (18.5):

```
010010.100 (18.5)            two's complement      010010.100 (18.5)
- 0110.110 (6.75)      ────────────────────────▶  +111001.010 (−6.75)
                            and sign extension      1001011.110 (11.75)

                            Carry bit is
                            discarded.
```

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 8-18 for more information.

Multiplication

The multiplication of two's complement fixed-point numbers is directly analogous to regular decimal multiplication, with the exception that the intermediate results must be sign extended so that their left sides align before you add them together.

For example, consider the multiplication of 10.11 (-1.25) with 011 (3):

The extra 1 is the result of necessary sign extension.

```
10.11 (−1.25)
 011 (3)
11011
1011
1100.01 (−3.75)
```

The number of fractional bits of the result is the sum of the number of fractional bits of the factors.

Multiplication Data Types

The following diagrams show the data types used for fixed-point multiplication in the Signal Processing Blockset. The diagrams illustrate the differences between the data types used for real-real, complex-real, and complex-complex multiplication. See individual reference pages in 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" on page 8-26, "Product Output Parameters" on page 8-25, and "Output Parameters" on page 8-28. These data types are defined in "Casts" on page 8-18.

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

Real-Real Multiplication. The following diagram shows the data types used in the multiplication of two real numbers in the Signal Processing Blockset. The output of this multiplication is in the product output data type:
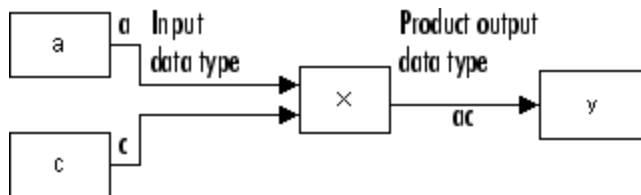
Real-Complex Multiplication. The following diagram shows the data types used in the multiplication of a real and a complex fixed-point number in the Signal Processing Blockset. Real-complex and complex-real multiplication are equivalent. The output of this multiplication is in the product output data type:
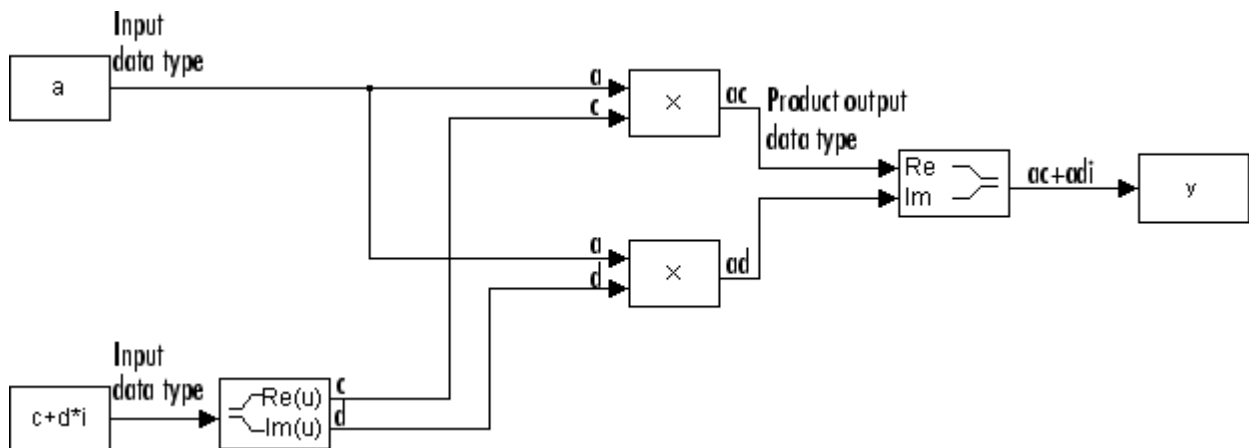
Complex-Complex Multiplication. The following diagram shows the multiplication of two complex fixed-point numbers in the Signal Processing Blockset. Note that the output of the multiplication is in the accumulator data type:
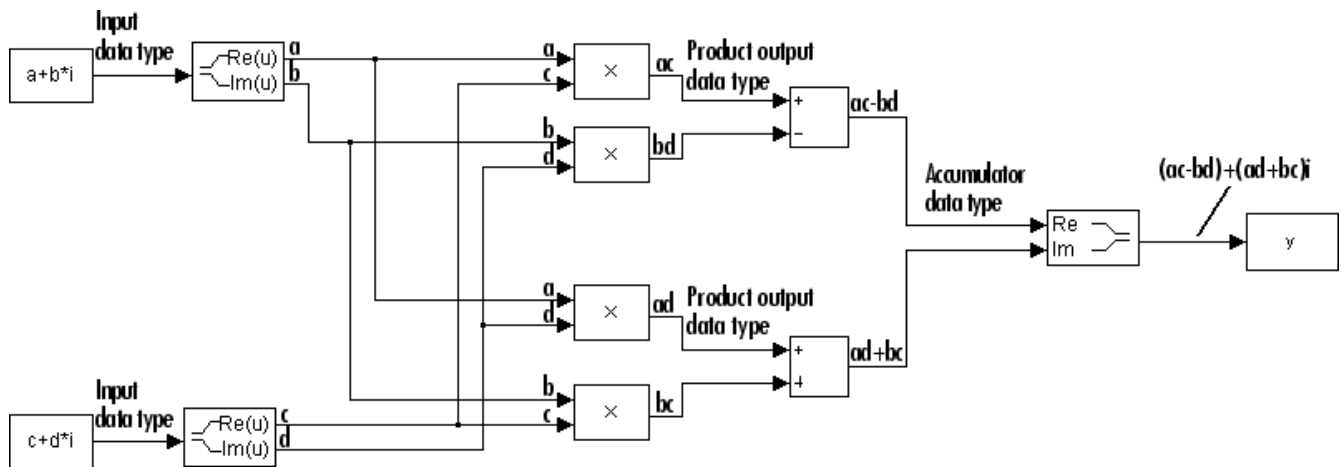
Casts

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, the addends 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" on page 8-26. 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" on page 8-24 and "Product Output Parameters" on page 8-25.

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 the Signal Processing Blockset 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:

The source bits must be shifted up to match the binary point position of the destination data type.

source

destination

This bit from the source data type "falls off" the high end with the shift up. Overflow can occur. The result will saturate or wrap.

These bits of the destination data type are padded with 0's or 1's.
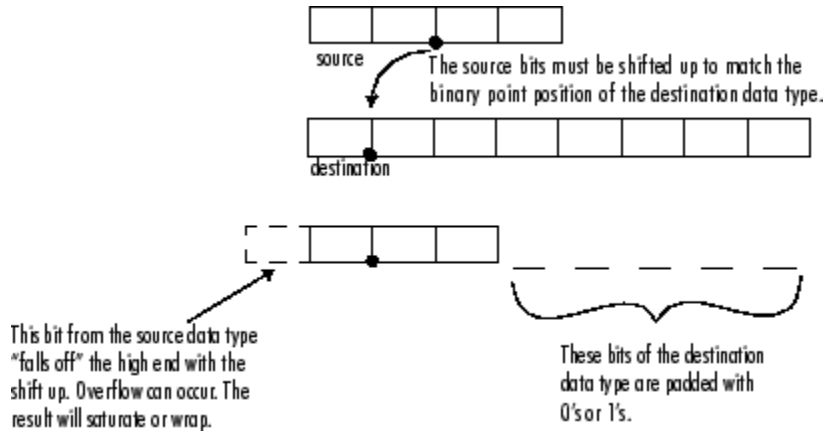
As the diagram shows, the source bits are shifted up so that the binary point matches the destination binary point position. The highest source bit does not fit, so overflow might occur and the result can saturate or wrap. The empty bits at the low end of the destination data type are padded with either 0's or 1's:

- If overflow does not occur, the empty bits are padded with 0's.

- If wrapping occurs, the empty bits are padded with 0's.

- If saturation occurs,

  - The empty bits of a positive number are padded with 1's.

  - The empty bits of a negative number are padded with 0's.

You can see that even with a cast from a shorter data type to a longer data type, overflow might still occur. This can happen when the integer length of the source data type (in this case two) is longer than the integer length of the destination data type (in this case one). Similarly, rounding might be necessary even when casting from a shorter data type to a longer data type, if the destination data type and scaling has fewer fractional bits than the source.

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:
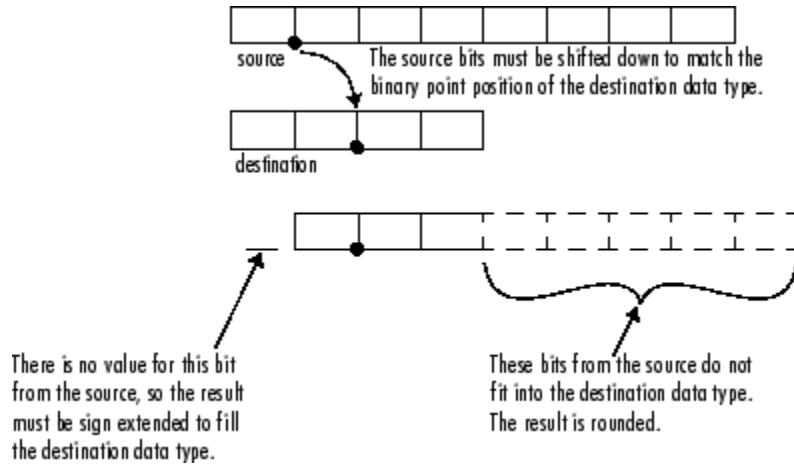
As the diagram shows, the source bits are shifted down so that the binary point matches the destination binary point position. There is no value for the highest bit from the source, so the result is sign extended to fill the integer portion of the destination data type. The bottom five bits of the source 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

The following sections describe how to set and monitor fixed-point settings for Signal Processing Blockset blocks both on a block-by-block and on a system-wide basis:

- "Setting Block Parameters" on page 8-22
- "Specifying System-Level Settings" on page 8-28

Setting Block Parameters

Blocks in the Signal Processing Blockset that have fixed-point support often allow you to specify fixed-point characteristics through block parameters. In many cases, such as with the accumulator and product output parameters, specifying these parameters enables you to simulate your target hardware more closely.

Note The fixed-point settings discussed in this section are ignored for floating-point signals.

Most fixed-point parameters for Signal Processing Blockset blocks appear when the **Fixed-point** tab is selected, for example on the Matrix Product block dialog below.
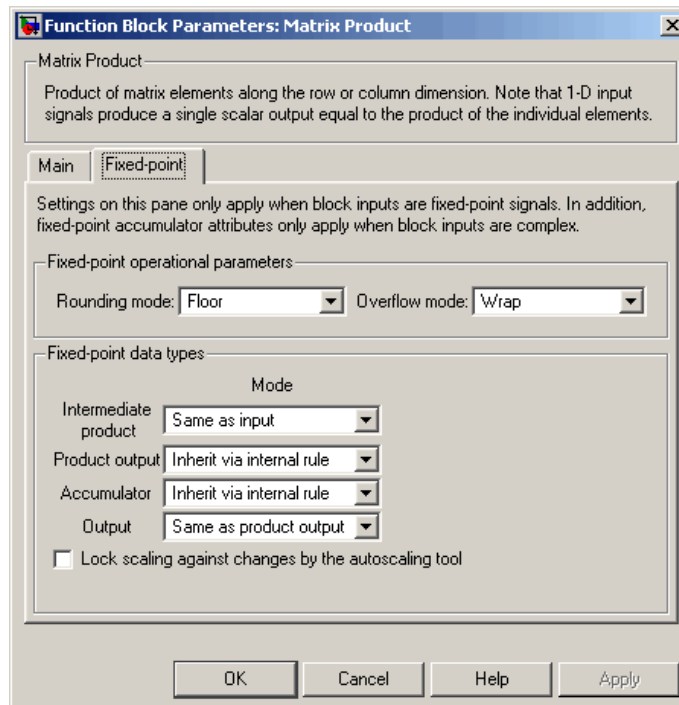
Many of the Signal Processing Blockset blocks with fixed-point capabilities share common parameters, though each block might have a different subset of these fixed-point parameters. The following parameters are discussed in this section:

- "Rounding Mode Parameter" on page 8-24
- "Overflow Mode Parameter" on page 8-24
- "Intermediate Product Parameters" on page 8-24
- "Product Output Parameters" on page 8-25
- "Accumulator Parameters" on page 8-26
- "Output Parameters" on page 8-28

For a discussion of all the parameters of a specific Signal Processing Blockset block, refer to the block's reference page in the Block Reference.

Remember that the Signal Processing Blockset does not allow mixed floating-point and fixed-point types on the input and output ports of its blocks. Therefore, the parameters discussed in this section only take effect if you feed the input port of that block with a fixed-point signal.

Rounding Mode Parameter

Use this parameter to specify the rounding method to be used when the result of a fixed-point calculation does not map exactly to a number representable by the data type and scaling that stores the result:

- `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, with the exact midpoint rounded to the closest representable number in the direction of positive infinity.

Overflow Mode Parameter

Use this parameter to specify the method to be used if the magnitude of a fixed-point calculation result does not fit into the range of the data type and scaling that stores the result:

- `Saturate` 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.

- `Wrap` uses modulo arithmetic to cast an overflow back into the representable range of the data type. See "Modulo Arithmetic" on page 8-13 for more information.

Intermediate Product Parameters

Fixed-point Signal Processing Blockset blocks that feed multiplication results back to the input of the multiplier usually allow you to specify the data type and scaling of the intermediate product:
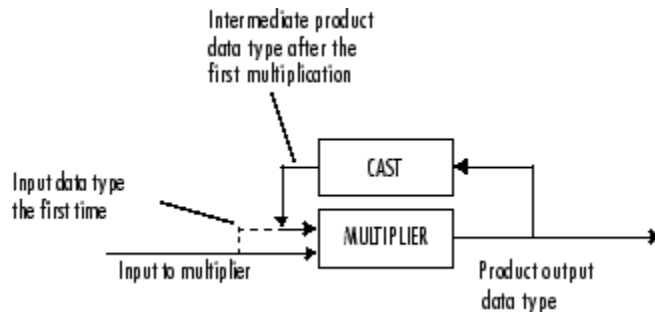
See the reference page of a specific block in to learn about the intermediate product data type for a specific block.

Use the **Intermediate product–Mode** parameter to specify how you would like to designate the intermediate product word and fraction lengths:

- When you select Same as input, these characteristics will match those of the first input to the block.

- When you select Binary point scaling, you are able to enter the word length and the fraction length of the intermediate product, in bits.

- When you select Slope and bias scaling, you are able to enter the word length, in bits, and the slope of the intermediate product. The bias of all signals in the Signal Processing Blockset is zero.

Product Output Parameters

Fixed-point Signal Processing Blockset blocks that must hold multiplication results for further calculation usually allow you to specify the data type and scaling of the product output:
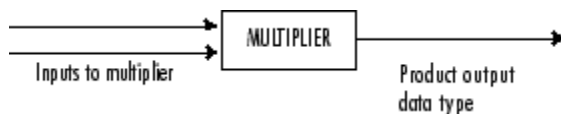
See the reference page of a specific block in to learn about the product output data type for a specific block. Note that for complex-complex multiplication, the multiplication result is in the accumulator data type. See "Multiplication

Data Types" on page 8-16 for more information on complex fixed-point multiplication in the Signal Processing Blockset.

Use the **Product output–Mode** parameter to specify how you would like to designate the product output word and fraction lengths:

- When you select Inherit via internal rule, the product output word and fraction lengths will be automatically calculated for you. In general, all the bits are preserved in the internal block algorithm for quantities using this selection. That is, the product output word and fraction lengths are selected such that

  - No overflow occurs
  - No precision loss occurs
  - Rounding modes have no effect

  Internal rule equations specific to each block are given in the block reference pages.

- When you select Same as input, these characteristics will match those of the first input to the block.

- When you select Binary point scaling, you are able to enter the word length and the fraction length of the product output, in bits.

- When you select Slope and bias scaling, you are able to enter the word length, in bits, and the slope of the product output. The bias of all signals in the Signal Processing Blockset is zero.

Accumulator Parameters

Fixed-point Signal Processing Blockset blocks that must hold summation results for further calculation usually allow you to specify the data type and scaling of the accumulator. Most such blocks cast to the accumulator data type prior to summation:

The result of each addition remains
in the accumulator data type.

Input(s) to adder → CAST → Accumulator data type → ADDER → Accumulator data type
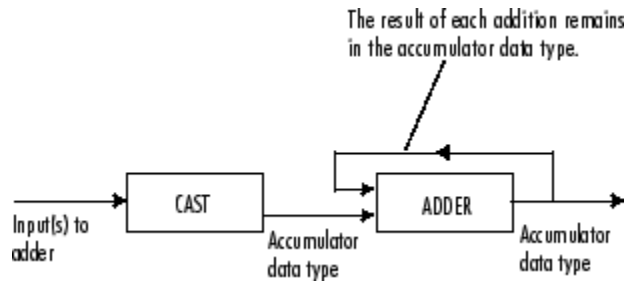
See the reference page of a specific block in for details on the accumulator data type of a specific block.

Use the **Accumulator–Mode** parameter to specify how you would like to designate the accumulator word and fraction lengths:

- When you select `Inherit via internal rule`, the accumulator output word and fraction lengths will be automatically calculated for you. In general, all the bits are preserved in the internal block algorithm for quantities using this selection. That is, the accumulator word and fraction lengths are selected such that

  - No overflow occurs

  - No precision loss occurs

  - Rounding modes have no effect

  Internal rule equations specific to each block are given in the block reference pages.

- When you select `Same as product output`, these characteristics will match those of the product output.

- When you select `Same as input`, these characteristics will match those of the first input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the accumulator, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the accumulator. The bias of all signals in the Signal Processing Blockset is zero.

Output Parameters

In many cases you can specify the output data type and scaling of fixed-point Signal Processing Blockset blocks.

Use the **Output–Mode** parameter to choose how you will specify the word length and fraction length of the output of the block:

- When you select `Same as accumulator`, these characteristics will match those of the accumulator.

- When you select `Same as product output`, these characteristics will match those of the product output.

- When you select `Same as input`, these characteristics will match those of the first input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the output, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the output. The bias of all signals in the Signal Processing Blockset is zero.

Specifying System-Level Settings

You can monitor and control fixed-point settings for Signal Processing Blockset blocks at a system or subsystem level via the Fixed-Point Settings interface.

Fixed-Point Settings Interface

Some fixed-point attributes of Signal Processing Blockset blocks can be monitored or set at the system level via the Fixed-Point Settings interface. For additional information on these subjects, see the following topics:

- The `fxptdlg` reference page — A reference page on the Fixed-Point Settings interface in the Simulink documentation

- "Tutorial: Feedback Controller Simulation" — A tutorial that highlights the use of the Fixed-Point Settings interface in the Simulink Fixed Point documentation

Logging. The Fixed-Point Settings interface logs overflows and saturations for fixed-point Signal Processing Blockset blocks. The Fixed-Point Settings interface does not log overflows and saturations when the `Data overflow` line in the **Diagnostics > Data Integrity** pane of the Configuration dialog is set to `None`.

The Fixed-Point Settings interface also logs the simulation minimums and maximums for certain fixed-point quantities of many Signal Processing Blockset blocks. The blocks that currently support logging of simulation minimums and maximums are:

- Autocorrelation
- Convolution
- Correlation
- Cumulative Product
- Cumulative Sum
- Difference
- Digital Filter
- FFT
- FIR Decimation
- FIR Interpolation
- FIR Rate Conversion
- IFFT
- Levinson-Durbin
- LMS Filter
- Magnitude FFT
- Matrix 1-Norm
- Matrix Product
- Matrix Scaling
- Matrix Sum

- Maximum

- Mean

- Median

- Minimum

- Normalization

- Two-Channel Analysis Subband Filter

- Two-Channel Synthesis Subband Filter

- Variance

- Window Function

The minimums and maximums of the following quantities are logged for the supported blocks:

- Product output(s)

- Accumulator(s)

- State

- Output(s)

- Stage input

- Stage output

- Tap sum

Autoscaling. You can use the Fixed-Point Settings interface autoscaling tool to set the scaling for the following Signal Processing Blockset fixed-point data types:

- Product output(s)

- Accumulator(s)

- State

- Output(s)

Note that autoscaling is only supported for blocks that log simulation minimums and maximums.

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 Settings interface. The `Scaled doubles` mode is also supported for Signal Processing Blockset source and byte-shuffling blocks that support [Slope Bias] signals, but not for arithmetic fixed-point Signal Processing Blockset blocks such as FFT or Digital Filter.

Fixed-Point Filtering

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 the Signal Processing Toolbox. 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.

Interoperability with Other Products

The following tables compare the supported features of various fixed-point products from The MathWorks:

Fixed-Point Data Type Support

	Signal Processing Blockset Fixed-Point	Filter Design Toolbox	Simulink Fixed-Point Blocks	Stateflow®
Custom floating-point	Partial support [1]	Yes	Yes (simulation) No (code generation)	No
Signed two's complement integer, fractional, and generalized fixed-point numbers	Yes	Yes	Yes	Yes

Fixed-Point Data Type Support (Continued)

	Signal Processing Blockset Fixed-Point	Filter Design Toolbox	Simulink Fixed-Point Blocks	Stateflow®
Unsigned integer, fractional, and generalized fixed-point numbers	Partial support [2]	Yes	Yes	Yes
Data type override	Partial support via the Fixed-Point Settings interface [3]	Yes, via the set function	Yes, via the Fixed-Point Settings interface	No

[1] Fixed-point Signal Processing Blockset blocks that only manipulate bits and do not perform arithmetic operations accept custom floating-point inputs. The source blocks Constant Diagonal Matrix and DSP Constant also allow you to specify a custom floating-point output data type.

[2] See the reference page of each fixed-point Signal Processing Blockset block to determine whether it supports unsigned fixed-point signals.

[3] 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 Settings interface. The Scaled doubles mode is also supported for Signal Processing Blockset source and byte-shuffling blocks that support [Slope Bias] signals, but not for other arithmetic fixed-point Signal Processing Blockset blocks.

Fixed-Point Scaling Support

	Signal Processing Blockset Fixed-Point	Filter Design Toolbox	Simulink Fixed-Point Blocks	Stateflow
[Slope Bias] scaling	Partial support [1]	No	Yes	Yes
Binary point-only scaling	Yes	Yes	Yes	Yes
Automatic scaling	Yes, via the Fixed-Point Settings interface	No	Yes, via the Fixed-Point Settings interface	No

[1] Fixed-point Signal Processing Blockset blocks that only manipulate bits and do not perform arithmetic operations accept [Slope Bias] signals with nonpower-of-two slope and nonzero bias. The following source blocks also allow you to specify a [Slope Bias] output signal: Constant Diagonal Matrix, Discrete Impulse, DSP Constant, Identity Matrix, and Sine Wave. Blocks that perform arithmetic operations require power-of-two slope and zero bias.

Fixed-Point Operations Support

	Signal Processing Blockset Fixed-Point	Filter Design Toolbox	Simulink Fixed-Point Blocks	Stateflow
Rounding methods	Floor, nearest	Ceiling, convergent, fix, floor, round	Ceiling, floor, nearest, zero	Offline conversions are rounded to nearest Online conversions are rounded to floor or zero
Overflow handling	Saturate, wrap	Saturate, wrap	Saturate, wrap	Simulation halts upon overflow
Logging	Yes, via the Fixed-Point Settings interface	No	Yes, via the Fixed-Point Settings interface	No

Fixed-Point Code Generation Support

	Signal Processing Blockset Fixed-Point	Filter Design Toolbox	Simulink Fixed-Point Blocks	Stateflow
C code generation	Yes	No	Yes	Yes

Building Models with Other Blocks

You can build models with fixed-point Signal Processing Blockset blocks that include fixed-point and floating-point blocks both from the Signal Processing Blockset and from other MathWorks products. The following sections discuss

issues to keep in mind when connecting fixed-point Signal Processing Blockset blocks to other types of blocks.

Connecting Fixed-Point and Floating-Point Blocks

Signal Processing Blockset blocks do not accept mixed floating-point and fixed-point types on their input and output ports. Therefore, if you want a Signal Processing Blockset block to have a fixed-point output data type, you must feed the block with a fixed-point input signal.

To feed a Signal Processing Blockset block with a fixed-point signal from another block that does not have fixed-point support, use the Simulink Data Type Conversion block, as in the model below:
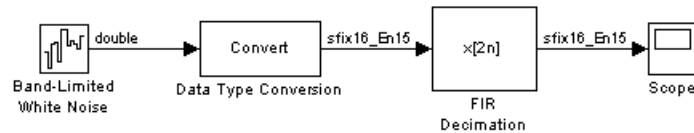
The Simulink Band-Limited White Noise block in the model does not allow you to set a fixed-point output data type and scaling in its block mask. The Data Type Conversion block, however, allows you to do so. The following shows the mask parameter settings of the Data Type Conversion block in the model:
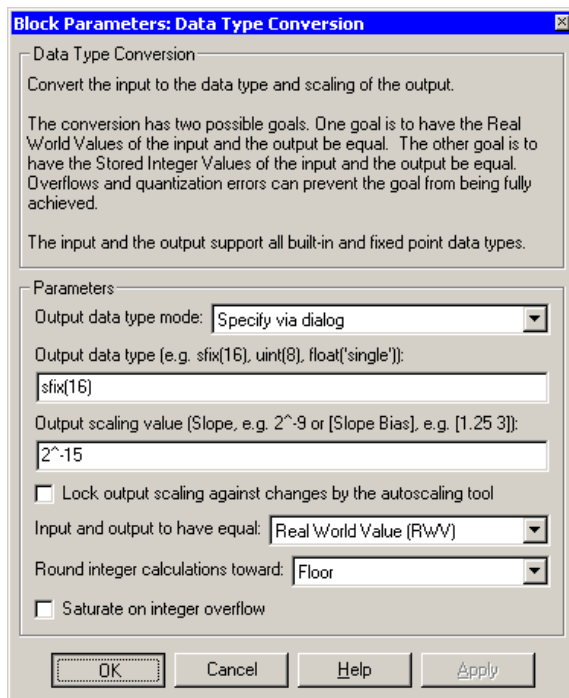
Note that the **Output scaling value** parameter of the Data Type Conversion block specifies a power-of-two scaling with 0 bias. This is a requirement for fixed-point signals in the Signal Processing Blockset, as discussed in the next section.

Similarly to the example above, you can feed the output of fixed-point Signal Processing Blockset blocks to other blocks that do not accept fixed-point data types by using the Data Type Conversion block.

Connecting Blocks with Different Scalings

Fixed-point signals in the Signal Processing Blockset must have a slope adjustment of 1 and a bias of 0; that is, only power-of-two or binary point-only scaling is accepted. You must make sure that any block that feeds the input port of a fixed-point Signal Processing Blockset block specifies binary point-only scaling for the output scaling of the block. Alternately, you can

use the Simulink Data Type Conversion block between any two fixed-point blocks of with different scalings.

9

Blocks — Categorical List

The Signal Processing Blockset contains the block libraries described in the following list. Access the libraries with the Simulink Library Browser, which you can open by typing `simulink`.

Note To find out about using blocks together for common signal processing tasks, see Chapter 1, "Working with Signals".

Select a library for a list of links to the online reference pages of its blocks. (For an alphabetical reference to block reference pages, see Chapter 10, "Blocks — Alphabetical List".)

Estimation

The following sublibraries reside in the Estimation library:

- "Linear Prediction" on page 9-2
- "Parametric Estimation" on page 9-3
- "Power Spectrum Estimation" on page 9-3

Linear Prediction

Blocks for linear prediction and working with linear prediction coefficients.

Autocorrelation LPC	Determine coefficients of Nth-order forward linear predictors
Levinson-Durbin	Solve linear system of equations using Levinson-Durbin recursion
LPC to LSF/LSP Conversion	Convert linear prediction coefficients (LPCs) to line spectral pairs (LSPs) or line spectral frequencies (LSFs)
LPC to/from Cepstral Coefficients	Convert linear prediction coefficients (LPCs) to cepstral coefficients (CCs) or cepstral coefficients to linear prediction coefficients
LPC to/from RC	Convert linear prediction coefficients (LPCs) to reflection coefficients (RCs) or reflection coefficients to linear prediction coefficients
LPC/RC to Autocorrelation	Convert linear prediction coefficients (LPCs) or reflection coefficients (RCs) to autocorrelation coefficients (ACs)
LSF/LSP to LPC Conversion	Convert line spectral frequencies (LSFs) or line spectral pairs (LSPs) to linear prediction coefficients (LPCs)

Parametric Estimation

Blocks for computing estimates of autoregressive model parameters using various methods.

Burg AR Estimator	Compute estimate of autoregressive (AR) model parameters using Burg method
Covariance AR Estimator	Compute estimate of autoregressive (AR) model parameters using covariance method
Modified Covariance AR Estimator	Compute estimate of autoregressive (AR) model parameters using modified covariance method
Yule-Walker AR Estimator	Compute estimate of autoregressive (AR) model parameters using Yule-Walker method

Power Spectrum Estimation

Blocks for computing parametric and nonparametric spectral estimates using various methods.

Burg Method	Compute parametric spectral estimate using Burg method
Covariance Method	Compute parametric spectral estimate using covariance method
Magnitude FFT	Compute nonparametric estimate of the spectrum using the periodogram method
Modified Covariance Method	Compute parametric spectral estimate using modified covariance method

Periodogram	Compute nonparametric estimate of the spectrum
Yule-Walker AR Estimator	Compute estimate of autoregressive (AR) model parameters using Yule-Walker method

Filtering

The following sublibraries reside in the Filtering library:

- "Adaptive Filters" on page 9-5
- "Filter Designs" on page 9-5
- "Multirate Filters" on page 9-6

Adaptive Filters

Blocks for computing filter estimates of an input using various algorithms.

Block LMS Filter	Compute filtered output, filter error, and filter weights for a given input and desired signal using Block LMS adaptive filter algorithm
Fast Block LMS Filter	Compute filtered output, filter error, and filter weights for a given input and desired signal using the Fast Block LMS adaptive filter algorithm
Kalman Adaptive Filter	Compute filter estimates for inputs using Kalman adaptive filter algorithm
LMS Filter	Compute filtered output, filter error, and filter weights for a given input and desired signal using LMS adaptive filter algorithm
RLS Filter	Compute filtered output, filter error, and filter weights for a given input and desired signal using RLS adaptive filter algorithm

Filter Designs

Blocks for designing and implementing various filters.

Analog Filter Design	Design and implement analog filters
Digital Filter	Filter each channel of input over time using static or time-varying digital filter implementations
Digital Filter Design	Design and implement digital FIR and IIR filters
Filter Realization Wizard	Construct filter realizations using the Digital Filter block or the Sum, Gain, and Delay blocks
Overlap-Add FFT Filter	Implement overlap-add method of frequency-domain filtering
Overlap-Save FFT Filter	Implement overlap-save method of frequency-domain filtering

Multirate Filters

Blocks for implementing various multirate filters.

CIC Decimation	Decimate signal using Cascaded Integrator-Comb filter
CIC Interpolation	Interpolate signal using Cascaded Integrator-Comb filter
Dyadic Analysis Filter Bank	Decompose signals into subbands with smaller bandwidths and slower sample rates
Dyadic Synthesis Filter Bank	Reconstruct signals from subbands with smaller bandwidths and slower sample rates
FIR Decimation	Filter and downsample input signals
FIR Interpolation	Upsample and filter input signals
FIR Rate Conversion	Upsample, filter, and downsample input signals

Two-Channel Analysis Subband Filter	Decompose signal into a high-frequency subband and a low-frequency subband
Two-Channel Synthesis Subband Filter	Reconstruct signal from a high-frequency subband and a low-frequency subband

Math Functions

The following sublibraries reside in the Math Functions library:

- "Math Operations" on page 9-8
- "Matrices and Linear Algebra" on page 9-8
- "Polynomial Functions" on page 9-11

Math Operations

Blocks for specialized math operations not provided in the Simulink math library.

Complex Exponential	Compute complex exponential function
Cumulative Product	Compute cumulative product of channel, column, or row elements
Cumulative Sum	Compute cumulative sum of channel, column, or row elements
dB Conversion	Convert magnitude data to decibels (dB or dBm)
dB Gain	Apply decibel gain
Difference	Compute element-to-element difference along rows or columns
Normalization	Normalize input by its 2-norm or squared 2-norm

Matrices and Linear Algebra

The following sublibraries reside in the Matrices and Linear Algebra sublibrary:

- "Linear System Solvers" on page 9-9
- "Matrix Factorizations" on page 9-9
- "Matrix Inverses" on page 9-10

• "Matrix Operations" on page 9-10

Linear System Solvers

Blocks that solve the matrix equation AX = B for X using various methods.

Backward Substitution	Solve $U\text{X}=B$ for X when U is an upper triangular matrix
Cholesky Solver	Solve $S\text{X}=B$ for X when S is square Hermitian positive definite matrix
Forward Substitution	Solve $L\text{X}=B$ for X when L is lower triangular matrix
LDL Solver	Solve $S\text{X}=B$ for X when S is square Hermitian positive definite matrix
Levinson-Durbin	Solve linear system of equations using Levinson-Durbin recursion
LU Solver	Solve $A\text{X}=B$ for X when A is square matrix
QR Solver	Find minimum-norm-residual solution to AX=B
SVD Solver	Solve AX=B using singular value decomposition

Matrix Factorizations

Blocks for factoring matrices using various methods.

Cholesky Factorization	Factor square Hermitian positive definite matrix into triangular components.
LDL Factorization	Factor square Hermitian positive definite matrices into lower, upper, and diagonal components
LU Factorization	Factor square matrix into lower and upper triangular components

QR Factorization	Factor rectangular matrix into unitary and upper triangular components
Singular Value Decomposition	Factor matrix using singular value decomposition

Matrix Inverses

Blocks for inverting matrices using various methods.

Cholesky Factorization	Factor square Hermitian positive definite matrix into triangular components.
LDL Inverse	Compute inverse of Hermitian positive definite matrix using LDL factorization
LU Inverse	Compute inverse of square matrix using LU factorization
Pseudoinverse	Compute Moore-Penrose pseudoinverse of matrix

Matrix Operations

Blocks for various matrix operations such as extracting the diagonal, overwriting matrix values, and multiplying matrices.

Constant Diagonal Matrix	Generate square, diagonal matrix
Create Diagonal Matrix	Create square diagonal matrix from diagonal elements
Extract Diagonal	Extract main diagonal of input matrix
Extract Triangular Matrix	Extract lower or upper triangle from input matrices
Identity Matrix	Generate matrix with ones on the main diagonal and zeros elsewhere

Matrix 1-Norm	Compute the 1-norm of a matrix
Matrix Exponential	Compute matrix exponential
Matrix Multiply	Multiply or divide inputs
Matrix Product	Multiply matrix elements along rows or columns
Matrix Scaling	Scale matrix rows or columns by specified vector
Matrix Square	Compute square of input matrix
Matrix Sum	Sum matrix elements along rows or columns
Overwrite Values	Overwrite submatrix or subdiagonal of input
Permute Matrix	Reorder matrix rows or columns
Reciprocal Condition	Compute reciprocal condition of square matrix in the 1-norm
Submatrix	Select subset of elements (submatrix) from matrix input
Toeplitz	Generate matrix with Toeplitz symmetry
Transpose	Compute matrix transpose

Polynomial Functions

Blocks for working with polynomials.

Least Squares Polynomial Fit	Compute polynomial coefficients that best fit input data in least squares sense
Polynomial Evaluation	Evaluate polynomial expression
Polynomial Stability Test	Use Schur-Cohn algorithm to determine whether all roots of input polynomial are inside unit circle

Platform-Specific I/O

Windows (WIN32)

Blocks for working with audio data in 32-bit Windows operating systems.

From Wave Device	Read audio data from standard audio device in real-time (32-bit Windows operating systems only)
From Wave File	Read audio data from Microsoft Wave (.wav) file
To Wave Device	Send audio data to standard audio device in real-time (32-bit Windows operating systems only)
To Wave File	Write audio data to file in Microsoft Wave (.wav) format (32-bit Windows operating systems only)

Quantizers

Blocks for quantizing data.

G711 Codec	Encode linear, pulse code modulation (PCM) narrowband speech signals using A-law or mu-law encoders. Decode index values into quantized output values using A-law or mu-law decoders. Convert between A-law and mu-law index values.
Scalar Quantizer	Convert an input signal into a set of quantized output values. Convert an input signal into a set of index values. Convert a set of index values into a quantized output signal.
Scalar Quantizer Decoder	Convert each index value into quantized output value
Scalar Quantizer Design	Start Scalar Quantizer Design Tool (SQDTool) to design scalar quantizer using Lloyd algorithm
Scalar Quantizer Encoder	Encode each input value by associating it with the index value of a quantization region
Uniform Decoder	Decode integer input into floating-point output
Uniform Encoder	Quantize and encode floating-point input into integer output
Vector Quantizer Decoder	Find vector quantizer codeword that corresponds to a given, zero-based index value

Vector Quantizer Design   Design vector quantizer using Vector
                                       Quantizer Design Tool (VQDTool)

Vector Quantizer Encoder   For a given input, find index
                                       of nearest codeword based on
                                       Euclidean or weighted Euclidean
                                       distance measure

Signal Management

The following sublibraries reside in the Signal Management library:

- "Buffers" on page 9-15
- "Indexing" on page 9-15
- "Signal Attributes" on page 9-16
- "Switches and Counters" on page 9-16

Buffers

Blocks for changing the sample rate or frame rate of a signal by accumulating input samples before outputting them.

Buffer	Buffer input sequence to smaller or larger frame size
Delay Line	Rebuffer sequence of inputs with one-sample shift
Queue	Store inputs in FIFO register
Stack	Store inputs into LIFO register.
Triggered Delay Line	Buffer sequence of inputs into frame-based output
Unbuffer	Unbuffer input frame into sequence of scalar outputs

Indexing

Blocks for manipulating the ordering of a signal such as selecting parts of a signal or flipping a signal.

Flip	Flip the input vertically or horizontally
Multiport Selector	Distribute arbitrary subsets of input rows or columns to multiple output ports

Overwrite Values	Overwrite submatrix or subdiagonal of input
Submatrix	Select subset of elements (submatrix) from matrix input
Variable Selector	Select subset of rows or columns from input

Signal Attributes

Blocks for inspecting or modifying signal attributes such as frame status and complexity.

Check Signal Attributes	Generate error when input signal does or does not match selected attributes exactly
Convert 1-D to 2-D	Reshape 1-D or 2-D input to a 2-D matrix with specified dimensions
Convert 2-D to 1-D	Convert 2-D matrix input to 1-D vector
Frame Conversion	Specify frame status of output signal
Inherit Complexity	Change complexity of input to match a reference signal

Switches and Counters

Blocks for performing an action when an event such as a threshold crossing in the data occurs.

Counter	Count up or down through specified range of numbers
Edge Detector	Detect transition from zero to a nonzero value
Event-Count Comparator	Detect threshold crossing of accumulated nonzero inputs

Multiphase Clock   Generate multiple binary clock
                                    signals

N-Sample Enable   Output ones or zeros for specified
                                    number of sample times

N-Sample Switch   Switch between two inputs after
                                    specified number of sample periods

Signal Operations

Blocks for performing operations on a signal.

Constant Ramp	Generate ramp signal with length based on input dimensions
Convolution	Compute convolution of two inputs
Delay	Delay discrete-time input by specified number of samples or frames
Downsample	Resample input at lower rate by deleting samples
Interpolation	Interpolate values of real input samples
Offset	Truncate vectors by removing or keeping beginning or ending values
Pad	Alter input dimensions by padding or truncating rows and/or columns
Peak Finder	Determine whether each value of input signal is local minimum or maximum
Repeat	Resample input at higher rate by repeating values
Sample and Hold	Sample and hold input signal
Triggered Signal From Workspace	Import signal samples from MATLAB workspace when triggered
Unwrap	Unwrap signal phase
Upsample	Resample input at higher rate by inserting zeros
Variable Fractional Delay	Delay input by time-varying fractional number of sample periods
Variable Integer Delay	Delay input by time-varying integer number of sample periods

Window Function	Compute and/or apply window to input signal
Zero Crossing	Count number of times signal crosses zero in a single time step
Zero Pad	Alter input dimensions by zero-padding (or truncating) rows and/or columns

Signal Processing Sinks

Various scopes and blocks for exporting signals to the MATLAB workspace.

Matrix Viewer	Display matrices as color images
Signal To Workspace	Write simulation data to array in MATLAB workspace
Spectrum Scope	Compute and display the periodogram of each input signal
To Multimedia File	Write video frames and/or audio samples to multimedia file
Triggered To Workspace	Write input sample to MATLAB workspace when triggered
Vector Scope	Display vector or matrix of time-domain, frequency-domain, or user-defined data
Waterfall	View vectors of data over time

Signal Processing Sources

Blocks that generate discrete-time signals such as sine waves and uniform random signals.

Chirp	Generate swept-frequency cosine (chirp) signal
Constant Diagonal Matrix	Generate square, diagonal matrix
Discrete Impulse	Generate discrete impulse
DSP Constant	Generate discrete- or continuous-time constant signal
From Multimedia File	Read video frames and/or audio samples from compressed multimedia file
Identity Matrix	Generate matrix with ones on the main diagonal and zeros elsewhere
Multiphase Clock	Generate multiple binary clock signals
N-Sample Enable	Output ones or zeros for specified number of sample times
Random Source	Generate randomly distributed values
Signal From Workspace	Import signal from MATLAB workspace
Sine Wave	Generate continuous or discrete sine wave

Statistics

Blocks for performing various statistical computations.

Autocorrelation	Compute autocorrelation of vector inputs
Correlation	Compute cross-correlation of two inputs
Detrend	Remove linear trend from vectors
Histogram	Generate histogram of input or sequence of inputs
Maximum	Find maximum values in an input or sequence of inputs
Mean	Find mean value of an input or sequence of inputs
Minimum	Find minimum values in an input or sequence of inputs
RMS	Compute root-mean-square (RMS) value of an input or sequence of inputs
Sort	Sort input elements by value
Standard Deviation	Find standard deviation of an input or sequence of inputs
Variance	Compute variance of an input or sequence of inputs

Transforms

Blocks for computing various transforms.

Analytic Signal	Compute analytic signals of discrete-time inputs
Complex Cepstrum	Compute complex cepstrum of input
DCT	Compute discrete cosine transform (DCT) of input
DWT	Compute discrete wavelet transform (DWT) of input
FFT	Compute fast Fourier transform (FFT) of input
IDCT	Compute inverse discrete cosine transform (IDCT) of input
IDWT	Compute inverse discrete wavelet transform (IDWT) of input
IFFT	Compute inverse fast Fourier transform (IFFT) of input
Inverse Short-Time FFT	Recover time-domain signals by performing an inverse short-time, fast Fourier transform (FFT)
Magnitude FFT	Compute nonparametric estimate of the spectrum using the periodogram method
Real Cepstrum	Compute real cepstrum of input
Short-Time FFT	Compute nonparametric estimate of the spectrum using short-time, fast Fourier transform (FFT) method

Blocks — Alphabetical List

Analog Filter Design

Purpose Design and implement analog filters

Library Filtering/ Filter Designs

Description
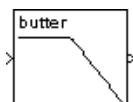
butter

The Analog Filter Design block designs and implements a Butterworth, Chebyshev type I, Chebyshev type II, or elliptic filter in a highpass, lowpass, bandpass, or bandstop configuration.

The input must be a sample-based scalar signal.

The design and band configuration of the filter are selected from the **Design method** and **Filter type** pop-up menus in the dialog box. For each combination of design method and band configuration, an appropriate set of secondary parameters is displayed.

Filter Design	Description
Butterworth	The magnitude response of a Butterworth filter is maximally flat in the passband and monotonic overall.
Chebyshev type I	The magnitude response of a Chebyshev type I filter is equiripple in the passband and monotonic in the stopband.
Chebyshev type II	The magnitude response of a Chebyshev type II filter is monotonic in the passband and equiripple in the stopband.
Elliptic	The magnitude response of an elliptic filter is equiripple in both the passband and the stopband.

The following table lists the available parameters for each design/band combination. For lowpass and highpass band configurations, these parameters include the passband edge frequency Ω_p, the stopband edge frequency Ω_s, the passband ripple R_p, and the stopband attenuation R_s. For bandpass and bandstop configurations, the parameters include the lower and upper passband edge frequencies, Ω_{p1} and Ω_{p2}, the lower and upper stopband edge frequencies, Ω_{s1} and Ω_{s2}, the passband ripple R_p, and the stopband attenuation R_s. Frequency values are in rad/s, and ripple and attenuation values are in dB.

	Lowpass	**Highpass**	**Bandpass**	**Bandstop**
`Butterworth`	Order, Ω_p	Order, Ω_p	Order, Ω_{p1}, Ω_{p2}	Order, Ω_{p1}, Ω_{p2}
`Chebyshev Type I`	Order, Ω_p, R_p	Order, Ω_p, R_p	Order, Ω_{p1}, Ω_{p2}, R_p	Order, Ω_{p1}, Ω_{p2}, R_p
`Chebyshev Type II`	Order, Ω_s, R_s	Order, Ω_s, R_s	Order, Ω_{s1}, Ω_{s2}, R_s	Order, Ω_{s1}, Ω_{s2}, R_s
`Elliptic`	Order, Ω_p, R_p, R_s	Order, Ω_p, R_p, R_s	Order, Ω_{p1}, Ω_{p2}, R_p, R_s	Order, Ω_{p1}, Ω_{p2}, R_p, R_s

The analog filters are designed using the Signal Processing Toolbox's filter design commands `buttap`, `cheb1ap`, `cheb2ap`, and `ellipap`, and are implemented in state-space form. Filters of order 8 or less are implemented in controller canonical form for improved efficiency.

Dialog Box
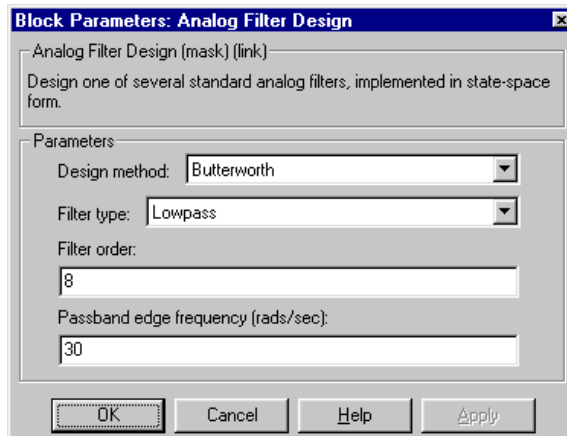
The parameters displayed in the dialog box vary for different design/band combinations. Only some of the parameters listed below are visible in the dialog box at any one time.

Analog Filter Design

Design method

The filter design method: `Butterworth`, `Chebyshev type I`, `Chebyshev type II`, or `Elliptic`. Tunable.

Filter type

The type of filter to design: `Lowpass`, `Highpass`, `Bandpass`, or `Bandstop`. Tunable.

Filter order

The order of the filter, for lowpass and highpass configurations. For bandpass and bandstop configurations, the order of the final filter is *twice* this value.

Passband edge frequency

The passband edge frequency, in rad/s, for the highpass and lowpass configurations of the Butterworth, Chebyshev type I, and elliptic designs. Tunable.

Lower passband edge frequency

The lower passband frequency, in rad/s, for the bandpass and bandstop configurations of the Butterworth, Chebyshev type I, and elliptic designs. Tunable.

Upper passband edge frequency

The upper passband frequency, in rad/s, for the bandpass and bandstop configurations of the Butterworth, Chebyshev type I, or elliptic designs. Tunable.

Stopband edge frequency

The stopband edge frequency, in rad/s, for the highpass and lowpass band configurations of the Chebyshev type II design. Tunable.

Lower stopband edge frequency

The lower stopband edge frequency, in rad/s, for the bandpass and bandstop configurations of the Chebyshev type II design. Tunable.

Upper stopband edge frequency

The upper stopband edge frequency, in rad/s, for the bandpass and bandstop filter configurations of the Chebyshev type II design. Tunable.

Passband ripple in dB

> The passband ripple, in dB, for the Chebyshev Type I and elliptic designs. Tunable.

Stopband attenuation in dB

> The stopband attenuation, in dB, for the Chebyshev Type II and elliptic designs. Tunable.

References Antoniou, A. *Digital Filters: Analysis, Design, and Applications*. 2nd ed. New York, NY: McGraw-Hill, 1993.

Supported Data Types

- Double-precision floating point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Digital Filter Design	Signal Processing Blockset
buttap	Signal Processing Toolbox
cheb1ap	Signal Processing Toolbox
cheb2ap	Signal Processing Toolbox
ellipap	Signal Processing Toolbox

See the following sections for related information:

- Chapter 3, "Filters"
- "Analog Filter Design Block" on page 3-51

Analytic Signal

Purpose　　　Compute analytic signals of discrete-time inputs

Library　　　Transforms

Description　　The Analytic Signal block computes the complex analytic signal corresponding to each channel of the real *M*-by-*N* input, *u*
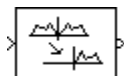
$$y = u + j\mathrm{H}\{u\}$$

where $j = \sqrt{-1}$ and H{ } denotes the Hilbert transform. The real part of the output in each channel is a replica of the real input in that channel; the imaginary part is the Hilbert transform of the input. In the frequency domain, the analytic signal retains the positive frequency content of the original signal while zeroing-out negative frequencies and doubling the DC component.

The block computes the Hilbert transform using an equiripple FIR filter with the order specified by the **Filter order** parameter, *n*. The linear phase filter is designed using the Remez exchange algorithm, and imposes a delay of *n*/2 on the input samples.

The output has the same dimension and frame status as the input.

Sample-Based Operation

When the input is sample based, each of the *M**N matrix elements represents an independent channel. Thus, the block computes the analytic signal for each channel (matrix element) over time.

Frame-Based Operation

When the input is frame based, each of the *N* columns in the matrix contains *M* sequential time samples from an independent channel, and the block computes the analytic signal for each channel over time.

Dialog Box
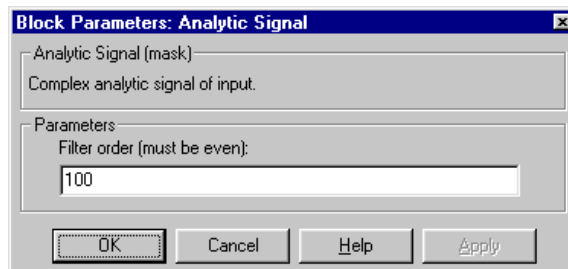
Filter order

The length of the FIR filter used to compute the Hilbert transform.

Supported Data Types

- Double-precision floating point

- Single-precision floating point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

Autocorrelation

Purpose   Compute autocorrelation of vector inputs

Library   Statistics

Description

The Autocorrelation block computes the autocorrelation of each channel in an input matrix or vector, u. The block computes the autocorrelation along each column of a frame-based input, and computes along the vector dimension of a sample-based vector input. The block does not accept sample-based matrix inputs. Outputs are always sample based.

M-by-N matrix inputs must be frame based. The result, y, is a sample-based $(l+1)$-by-N matrix whose jth column has elements

$$y_{i,j} = \sum_{k=1}^{M} u_{k,j}^{*} u_{(k+i-1),j} \qquad 1 \le i \le (l+1)$$

where * denotes the complex conjugate, and l represents the maximum lag. Note that y_1j is the zero-lag element in the jth column. When you select **Compute all non-negative lags**, $l=M$. Otherwise, l is specified as a nonnegative integer by the **Maximum non-negative lag (less than input length)** parameter.

Input u is zero when indexed outside of its valid range. When the input is real, the output is real; otherwise, the output is complex.

When the input is a sample-based vector (row, column, or 1-D), the output is sample based, with the same shape as the input and length $l+1$. The block computes the autocorrelation of sample-based vector inputs along the vector dimensions. The Autocorrelation block does not accept a sample-based full-dimension matrix input.

The Autocorrelation block accepts both real and complex fixed-point and floating-point inputs. Fixed-point signals are not supported for the frequency domain.

Fixed-Point Data Types

The following diagram shows the data types used within the Autocorrelation block for fixed-point signals (time domain only).
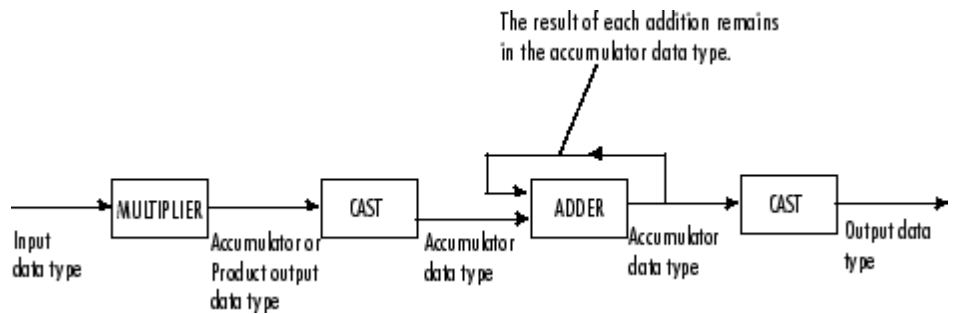
The result of each addition remains
in the accumulator data type.

You can set the product output, accumulator, and output data types in
the block dialog as discussed below.

The output of the multiplier is in the product output data type
when the input is real. When the input is complex, the result of the
multiplication is in the accumulator data type. For details on the
complex multiplication performed, refer to "Multiplication Data Types"
on page 8-16.

Autocorrelation

Dialog Box

The **Main** pane of the Autocorrelation block dialog appears as follows:
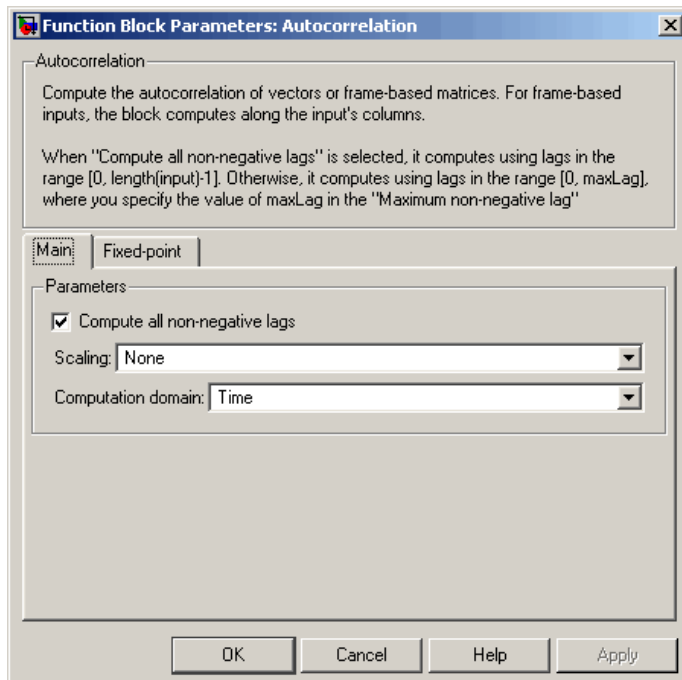
Compute all non-negative lags

Select to compute the autocorrelation over all nonnegative lags in the range [0, length(input)-1].

Maximum non-negative lag (less than input length)

Specify the maximum positive lag, l, for the autocorrelation. This parameter is enabled when you do not select the **Compute all non-negative lags** check box.

Scaling

This parameter controls the scaling that is applied to the output. The following options are available:

- None — Generates the raw autocorrelation, $y_{i,j}$, without normalization

- Biased — Generates the biased estimate of the autocorrelation

$$y_{i,j}^{biased} = \frac{y_{i,j}}{M}$$

- Unbiased — Generates the unbiased estimate of the autocorrelation

$$y_{i,j}^{unbiased} = \frac{y_{i,j}}{M-i}$$

- Unity at zero-lag — Normalizes the estimate of the autocorrelation for each channel so that the zero-lag sum is identically 1

$$y_{1,j} = 1$$

Note The **Scaling** parameter must be set to None for fixed-point signals.

This parameter is tunable, except in the Simulink external mode.

Computation domain

This parameter sets the domain in which the block computes convolutions to one of the following settings:

- Time — Computes in the time domain, which minimizes memory use

- Frequency — Computes in the frequency domain, which might require fewer computations than computing in the time domain, depending on the input length

Autocorrelation

The **Fixed-point** pane of the Autocorrelation block dialog appears as follows:

Rounding mode

Select the rounding mode for fixed-point operations.

Overflow mode

Select the overflow mode for fixed-point operations.

Product output

Use this parameter to specify how you would like to designate the product output word and fraction lengths. Refer to "Fixed-Point Data Types" on page 10-8 and "Multiplication Data Types" on page 8-16 for illustrations depicting the use of the product output data type in this block:

- When you select `Inherit via internal rule`, the product output word length and fraction length are automatically set according to the following equations:

$$ideal\ product\ output\ word\ length\ =\ 2 \times input\ word\ length$$

$$ideal\ product\ output\ fraction\ length\ =\ 2 \times input\ fraction\ length$$

> **Note** The actual product output word length may be equal to or greater than the calculated ideal product output word length, depending on the settings on the **Hardware Implementation** pane of the Configuration Parameters dialog box.

- When you select `Same as input`, these characteristics will match those of the input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the product output, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the product output. This block requires power-of-two slope and a bias of zero.

Accumulator

The result of each addition remains in the accumulator data type.

Input to adder - accumulator or product output data type → CAST → Accumulator data type → ADDER → Accumulator data type

As depicted above, inputs to the accumulator are cast to the accumulator data type. The output of the adder remains in the accumulator data type as each element of the input is added to it. Use this parameter to specify how you would like to designate this accumulator word and fraction lengths.

You also use this parameter to specify the accumulator word and fraction lengths resulting from a complex-complex multiplication in the block. Refer to "Multiplication Data Types" on page 8-16 for more information.

- When you select Inherit via internal rule, the accumulator word length and fraction length are automatically set according to the following equations:

  If the input is real:

  $$ideal\ accumulator\ word\ length = ideal\ product\ output\ word\ length$$
  $$+\ floor(\log 2(number\ of\ samples\ per\ channel - 1)) + 1$$

  $$ideal\ accumulator\ fraction\ length =$$
  $$ideal\ product\ output\ fraction\ length$$

  If the input is complex:

  $$ideal\ accumulator\ word\ length = ideal\ product\ output\ word\ length$$
  $$+\ floor(\log 2(number\ of\ samples\ per\ channel - 1)) + 2$$

$$ideal\ accumulator\ fraction\ length\ =$$
$$ideal\ product\ output\ fraction\ length$$

Note The actual accumulator word length may be equal to or greater than the calculated ideal product output word length, depending on the settings on the **Hardware Implementation** pane of the Configuration Parameters dialog box.

- When you select Same as product output, these characteristics will match those of the product output.

- When you select Same as input, these characteristics will match those of the input to the block.

- When you select Binary point scaling, you are able to enter the word length and the fraction length of the accumulator, in bits.

- When you select Slope and bias scaling, you are able to enter the word length, in bits, and the slope of the accumulator. This block requires power-of-two slope and a bias of zero.

Output

Choose how you will specify the output word length and fraction length:

- When you select Same as accumulator, these characteristics will match those of the accumulator.

    A special case occurs when Inherit via internal rule is specified for **Accumulator**, and the block input is complex. In that case, the output word length will be one less than the accumulator word length.

- When you select Same as product output, these characteristics will match those of the product output.

- When you select Same as input, these characteristics will match those of the input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the output, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the output. This block requires power-of-two slope and a bias of zero.

Lock scaling against changes by the autoscaling tool
Select this parameter to prevent any fixed-point scaling you specify in this block mask from being overriden by the autoscaling tool in the Fixed-Point Settings interface. For more information about the autoscaling tool, refer to "Fixed-Point Settings Interface" on page 8-28.

Supported Data Types

- Double-precision floating point

- Single-precision floating point

- Fixed point (signed only)

- 8-, 16-, and 32-bit signed integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

| Correlation | Signal Processing Blockset |
| xcorr | Signal Processing Toolbox |

Purpose   Determine coefficients of Nth-order forward linear predictors

Library   Estimation / Linear Prediction
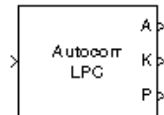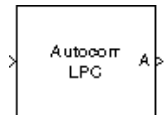
Description   The Autocorrelation LPC block determines the coefficients of an *N-step forward linear predictor* for the time-series in length-*M* input vector, *u*, by minimizing the prediction error in the least squares sense. A linear predictor is an FIR filter that predicts the next value in a sequence from the present and past inputs. This technique has applications in filter design, speech coding, spectral analysis, and system identification.

The Autocorrelation LPC block can output the prediction error as polynomial coefficients, reflection coefficients, or both. It can also output the prediction error power. The length-*M* input, *u*, can be a scalar, 1-D vector, frame- or sample-based column vector, or a sample-based row vector. Frame-based row vectors are not valid inputs.

When you select **Inherit prediction order from input dimensions**, the prediction order, *N*, is inherited from the input dimensions. Otherwise, you can use the **Prediction order** parameter to specify the value of *N*. Note that *N* must be less than the length of the input vector or the block produces an error.

When **Output(s)** is set to A, port A is enabled. Port A outputs an $(N+1)$-by-1 column vector, $a = [1\ a_2\ a_3\ ...\ a_{N+1}]^T$, containing the coefficients of an Nth-order moving average (MA) linear process that predicts the next value, $\hat{u}_{M+1}$, in the input time-series.

$$\hat{u}_{M+1} = -(a_2 u_M) - (a_3 u_{M-1}) - ... - (a_{N+1} u_{M-N+1})$$

When **Output(s)** is set to K, port K is enabled. Port K outputs a length-*N* column vector whose elements are the prediction error reflection coefficients. When **Output(s)** is set to A and K, both port A and K are enabled, and each port outputs its respective column vector of prediction coefficients. The outputs at both port A and K are always 1-D vectors.

When you select **Output prediction error power (P)**, port P is enabled. The prediction error power, a scalar, is output at port P.

Autocorrelation LPC

Algorithm

The Autocorrelation LPC block computes the least squares solution to

$$\min_{\tilde{a} \in \Re^n} \|U\tilde{a} - b\|$$

where $\|\cdot\|$ indicates the 2-norm and

$$U = \begin{bmatrix} u_1 & 0 & \cdots & 0 \\ u_2 & u_1 & \ddots & \vdots \\ \vdots & u_2 & \ddots & 0 \\ \vdots & \vdots & \ddots & u_1 \\ \vdots & \vdots & \vdots & u_2 \\ \vdots & \vdots & \vdots & \vdots \\ u_M & \vdots & \vdots & \vdots \\ 0 & \ddots & \vdots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & u_M \end{bmatrix}, \quad \tilde{a} = \begin{bmatrix} a_2 \\ \vdots \\ a_{n+1} \end{bmatrix}, \quad b = \begin{bmatrix} u_2 \\ u_3 \\ \vdots \\ u_M \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

Solving the least squares problem via the normal equations

$$U^* U\tilde{a} = U^* b$$

leads to the system of equations

$$\begin{bmatrix} r_1 & r_2^* & \cdots & r_n^* \\ r_2 & r_1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & r_2^* \\ r_n & \cdots & r_2 & r_1 \end{bmatrix} \begin{bmatrix} a_2 \\ a_3 \\ \vdots \\ a_{n+1} \end{bmatrix} = \begin{bmatrix} -r_2 \\ -r_3 \\ \vdots \\ -r_{n+1} \end{bmatrix}$$

where $r = [r_1\ r_2\ r_3 \dots r_{n+1}]^T$ is an autocorrelation estimate for u computed using the Autocorrelation block, and * indicates the complex conjugate transpose. The normal equations are solved in $O(n^2)$ operations by the Levinson-Durbin block.

Note that the solution to the LPC problem is very closely related to the Yule-Walker AR method of spectral estimation. In that context, the normal equations above are referred to as the Yule-Walker AR equations.

Dialog Box
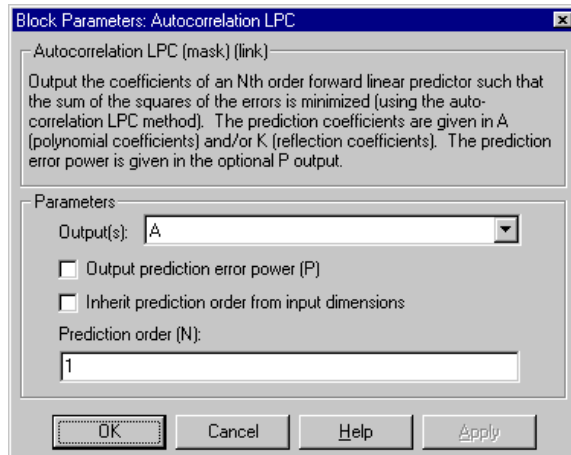
Output(s)

The type of prediction coefficients output by the block. The block can output polynomial coefficients (`A`), reflection coefficients (`K`), or both (`A and K`).

Output prediction error power (P)

When selected, enables port `P`, which outputs the output prediction error power.

Inherit prediction order from input dimensions

When selected, the block inherits the prediction order from the input dimensions.

Prediction order (N)

The prediction order, *N*. This parameter is disabled when you select the **Inherit prediction order from input dimensions** parameter.

Autocorrelation LPC

References

Haykin, S. *Adaptive Filter Theory*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1996.

Ljung, L. *System Identification: Theory for the User*. Englewood Cliffs, NJ: Prentice Hall, 1987. Pgs. 278-280.

Proakis, J. and D. Manolakis. *Digital Signal Processing*. 3rd ed. Englewood Cliffs, NJ: Prentice-Hall, 1996.

Supported Data Types

- Double-precision floating point
- Single-precision floating point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Autocorrelation	Signal Processing Blockset
Levinson-Durbin	Signal Processing Blockset
Yule-Walker Method	Signal Processing Blockset
lpc	Signal Processing Toolbox

Purpose   Solve *U*X=*B* for *X* when *U* is an upper triangular matrix

Library   Math Functions / Matrices and Linear Algebra / Linear System Solvers
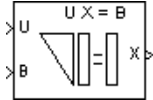
Description   The Backward Substitution block solves the linear system *U*X=*B*
by simple backward substitution of variables, where *U* is the upper
triangular *M*-by-*M* matrix input to the U port, and *B* is the *M*-by-*N*
matrix input to the B port. The output is the solution of the equations,
the *M*-by-*N* matrix X, and is always sample based. The block does not
check the rank of the inputs.

The block uses only the elements in the *upper triangle* of input *U*; the
lower elements are ignored. When you select the **Force input to be
unit-upper triangular** check box, the block replaces the elements on
the diagonal of *U* with 1's. This is useful when matrix *U* is the result
of another operation, such as an LDL decomposition, that uses the
diagonal elements to represent the D matrix.

A length-*M* vector input at port B is treated as an *M*-by-1 matrix.

**Dialog
Box**

Block Parameters: Backward Substitution

─ Backward Substitution (mask) ─

Solve UX=B where U is an upper (or unit-upper) triangular matrix. U must
be square. B must have the same number of rows as U.

─ Parameters ─

☐ Force input to be unit-upper triangular

| OK | Cancel | Help | Apply |
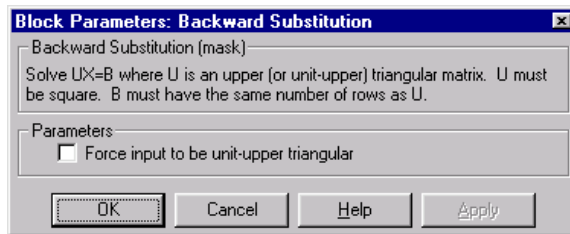
Force input to be unit-upper triangular
Replaces the elements on the diagonal of U with 1's when selected.
Tunable.

Backward Substitution

Supported Data Types

- Double-precision floating point

- Single-precision floating point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Cholesky Solver	Signal Processing Blockset
Forward Substitution	Signal Processing Blockset
LDL Solver	Signal Processing Blockset
Levinson-Durbin	Signal Processing Blockset
LU Solver	Signal Processing Blockset
QR Solver	Signal Processing Blockset

See "Solving Linear Systems" on page 6-7 for related information.

Purpose   Apply a cascade of biquadratic (second-order section) filters to the input.

Library   dspobslib

Description
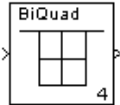
The Biquadratic Filter block applies a cascade of biquadratic filters independently to each input channel. Biquadratic filters are useful for reduced precision implementations because the coefficients are bounded between ±2 for typical minimum-phase designs. This may reduce scaling and coefficient sensitivity problems.

The filter is constructed from L second-order sections, each having a quadratic numerator and denominator.

$$H(z) = \prod_{k=1}^{L} H_k(z) = \prod_{k=1}^{L} \frac{b_{1k} + b_{2k}z^{-1} + b_{3k}z^{-2}}{a_{1k} + a_{2k}z^{-1} + a_{3k}z^{-2}}$$

The figure below illustrates the structure of a 4th-order biquadratic filter (L=2) with states $v$ik, where k is the section number.

Biquadratic Filter
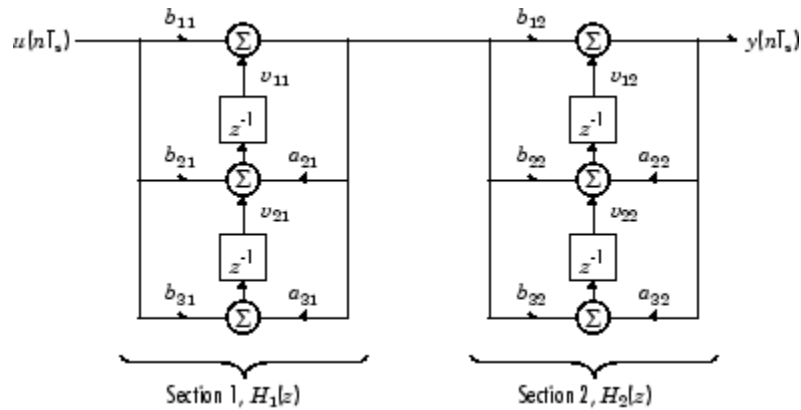
An *M*-by-*N* sample-based matrix input is treated as *M*N* independent channels, and an *M*-by-*N* frame-based matrix input is treated as *N* independent channels. In both cases, the block filters each channel independently over time, and the output has the same size and frame status as the input.

The **SOS matrix** parameter specifies the filter coefficients as a second-order section matrix of the type produced by the ss2sos and tf2sos functions in the Signal Processing Toolbox.

$$\begin{bmatrix} b_{11} & b_{21} & b_{31} & a_{11} & a_{21} & a_{31} \\ b_{12} & b_{22} & b_{32} & a_{12} & a_{22} & a_{32} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ b_{1L} & b_{2L} & b_{3L} & a_{1L} & a_{2L} & a_{3L} \end{bmatrix}$$

$$a_{11} = a_{12} = ... = a_{1L}$$

This is an *L*-by-6 matrix whose rows contain the numerator and denominator coefficients $b_i k$ and $a_i k$ of each second-order section in $H(z)$. Use the ss2sos and tf2sos functions to convert a state-space or transfer-function description of the filter into the second-order section description used by this block. Note that the filter uses a value of 1 for

the zero-delay denominator coefficients (a_{11} to a_{1L}) regardless of the value specified in the **SOS matrix** parameter.

The **Initial conditions** parameter sets the initial filter states, and can be specified in the following different forms:

- *Scalar* to be used for all filter states ($v_{11}, v_{12}, ..., v_{1L}, v_{21} v_{22}, ..., v_{2L}$) in all channels. An empty vector, [ ], is the same as the scalar value 0.

- *Vector* of length 2*L (row or column) to initialize the filter states for all channels.

$$\underbrace{\begin{bmatrix} v_{11}\ v_{21} \end{bmatrix}}_{H_1[z]}\ \underbrace{v_{12}\ v_{22}}_{H_2[z]}\ \cdots\ \underbrace{v_{1L}\ v_{2L}}_{H_L[z]}$$

Each pair of elements specifies v_1k and v_2k for second-order section k in every channel.

- *Matrix* of dimension (2*L)-by-N containing the initial filter states for each of the N channels.

$$
\begin{aligned}
H_1[z] &\left\{ \begin{bmatrix} v_{11}^{ch\,1} & v_{11}^{ch\,2} & \cdots & v_{11}^{ch\,N} \\ v_{21}^{ch\,1} & v_{21}^{ch\,2} & \cdots & v_{21}^{ch\,N} \end{bmatrix} \right. \\
H_2[z] &\left\{ \begin{bmatrix} v_{12}^{ch\,1} & v_{12}^{ch\,2} & \cdots & v_{12}^{ch\,N} \\ v_{22}^{ch\,1} & v_{22}^{ch\,2} & \cdots & v_{22}^{ch\,N} \end{bmatrix} \right. \\
&\quad \vdots \quad\ \vdots \quad\ \ddots \quad\ \vdots \\
H_L[z] &\left\{ \begin{bmatrix} v_{1L}^{ch\,1} & v_{1L}^{ch\,2} & \cdots & v_{1L}^{ch\,N} \\ v_{2L}^{ch\,1} & v_{2L}^{ch\,2} & \cdots & v_{2L}^{ch\,N} \end{bmatrix} \right.
\end{aligned}
$$

Each pair of elements in a *column* specifies v_1k and v_2k for second-order section k of the corresponding channel.

Biquadratic Filter

Dialog Box
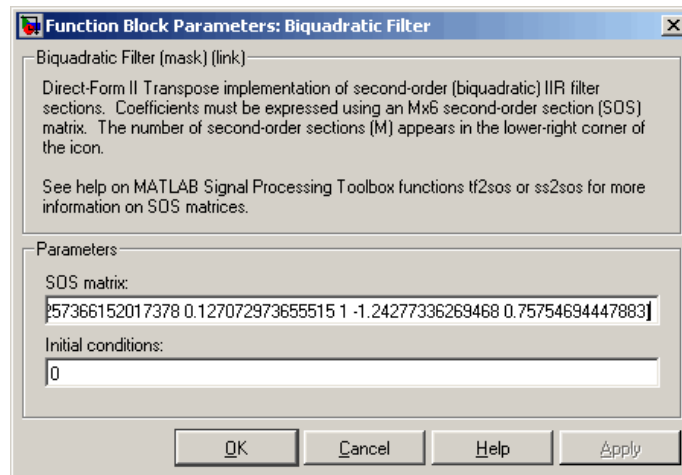
SOS matrix

> The second-order section matrix specifying the filter's coefficients. This matrix can be generated from state-space or transfer-function descriptions by using the Signal Processing Toolbox functions `ss2sos` and `tf2sos`.

Initial conditions

> The filter's initial conditions, a scalar, vector, or matrix.

Purpose   Compute filtered output, filter error, and filter weights for a given input and desired signal using Block LMS adaptive filter algorithm

Library   Filtering / Adaptive Filters

Description
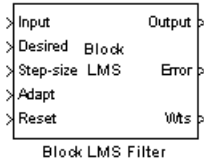
Block LMS Filter

The Block LMS Filter block implements an adaptive least mean-square (LMS) filter, where the adaptation of filter weights occurs once for every block of samples. The block estimates the filter weights, or coefficients, needed to minimize the error, $e(n)$, between the output signal, $y(n)$, and the desired signal, $d(n)$. Connect the signal you want to filter to the Input port. This input signal can be a sample-based scalar or a single-channel frame-based signal. Connect the signal you want to model to the Desired port. The desired signal must have the same data type, frame status, complexity, and dimensions as the input signal. The Output port outputs the filtered input signal, which can be sample or frame based. The Error port outputs the result of subtracting the output signal from the desired signal.

The block calculates the filter weights using the Block LMS adaptive filter algorithm. This algorithm is defined by the following equations.

$$n = kN + i$$
$$y(n) = \mathbf{w}^T(k-1)\mathbf{u}(n)$$
$$e(n) = d(n) - y(n)$$
$$\mathbf{w}(k) = \mathbf{w}(k-1) + f(\mathbf{u}(n), e(n), \mu)$$

The weight update function for the Block LMS adaptive filter algorithm is defined as

$$f(\mathbf{u}(n), e(n), \mu) = \mu \sum_{i=0}^{N-1} \mathbf{u}^*(kN+i)e(kN+i)$$

The variables are as follows.

Block LMS Filter

Variable	Description
n	The current time index
i	The iteration variable in each block, $0 \leq i \leq N-1$
k	The block number
N	The block size
$\mathbf{u}(n)$	The vector of buffered input samples at step *n*
$\mathbf{w}(n)$	The vector of filter-tap estimates at step *n*
y(n)	The filtered output at step *n*
e(*n*)	The estimation error at time *n*
d(*n*)	The desired response at time *n*
μ	The adaptation step size

Use the **Filter length** parameter to specify the length of the filter weights vector.

The **Block size** parameter determines how many samples of the input signal are acquired before the filter weights are updated. The input frame length must be a multiple of the **Block size** parameter.

The adaptation **Step-size (mu)** parameter corresponds to μ in the equations. You can either specify a step-size using the input port, Step-size, or enter a value in the Block Parameters: Block LMS Filter dialog box.

Use the **Leakage factor (0 to 1)** parameter to specify the leakage factor, $0 < 1 - \mu\alpha \leq 1$, in the leaky LMS algorithm shown below.

$$\mathbf{w}(k) = (1 - \mu\alpha)\mathbf{w}(k-1) + f(\mathbf{u}(n), e(n), \mu)$$

Enter the initial filter weights as a vector or a scalar in the **Initial value of filter weights** text box. When you enter a scalar, the block uses the scalar value to create a vector of filter weights. This vector

has length equal to the filter length and all of its values are equal to the scalar value

When you select the **Adapt port** check box, an Adapt port appears on the block. When the input to this port is greater than zero, the block continuously updates the filter weights. When the input to this port is zero, the filter weights remain at their current values.

When you want to reset the value of the filter weights to their initial values, use the **Reset input** parameter. The block resets the filter weights whenever a reset event is detected at the Reset port. The reset signal rate must be the same rate as the data signal input.

From the **Reset input** list, select None to disable the Reset port. To enable the Reset port, select one of the following from the **Reset input** list:

- Rising edge — Triggers a reset operation when the Reset input does one of the following:

  - Rises from a negative value to a positive value or zero

  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure).

- Falling edge — Triggers a reset operation when the Reset input does one of the following:

  - Falls from a positive value to a negative value or zero

- Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero (see the following figure)

- **Either edge** — Triggers a reset operation when the Reset input is a Rising edge or Falling edge (as described above)

- **Non-zero sample** — Triggers a reset operation at each sample time that the Reset input is not zero

Note When running simulations in the Simulink MultiTasking mode, sample-based reset signals have a one-sample latency, and frame-based reset signals have one frame of latency. Thus, there is a one-sample or one-frame delay between the time the block detects a reset event, and when it applies the reset. For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" on page 2-56 and "Models with Multiple Sample Rates" in the Real-Time Workshop User's Guide documentation.

Select the **Output filter weights** check box to create a Wts port on the block. For each iteration, the block outputs the current updated filter weights from this port.

Dialog Box

Block Parameters: Block LMS Filter

Block LMS Filter (mask) (link)

Computes filter weights based on the Block LMS algorithm for filtering of the input signal. The filter weights are updated once for every block of data that is processed.

Select the Adapt port check box to create an Adapt port on the block. When the input to this port is nonzero, the block continuously updates the filter weights. When the input to this port is zero, the filter weights remain constant.

If the Reset port is enabled and a reset event occurs, the block resets the filter weights to their initial values.

Parameters

Filter length:
32

Block size:
32

Specify step size via: Dialog

Step size (mu):
0.1

Leakage factor (0 to 1):
1.0

Initial value of filter weights:
0

☐ Adapt port

Reset port: None

☑ Output filter weights

[OK] [Cancel] [Help] [Apply]

Filter length

Enter the length of the FIR filter weights vector.

Block size

Enter the number of samples to acquire before the filter weights are updated. The input frame length must be an integer multiple of the block size.

Specify step-size via
> Select Dialog to enter a value for mu in the Block parameters:
> LMS Filter dialog box. Select Input port to specify mu using
> the Step-size input port.

Step-size (mu)
> Enter the step-size. Tunable.

Leakage factor (0 to 1)
> Enter the leakage factor, $0 < 1 - \mu\alpha \leq 1$. Tunable.

Initial value of filter weights
> Specify the initial values of the FIR filter weights.

Adapt port
> Select this check box to enable the Adapt input port.

Reset input
> Select this check box to enable the Reset input port.

Output filter weights
> Select this check box to export the filter weights from the Wts port.

References Hayes, M. H. *Statistical Digital Signal Processing and Modeling*. New York: John Wiley & Sons, 1996.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
Desired	• Double-precision floating point
	• Single-precision floating point
Step-size	• Double-precision floating point
	• Single-precision floating point

Port	Supported Data Types
Adapt	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Reset	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point
Error	• Double-precision floating point • Single-precision floating point
Wts	• Double-precision floating point • Single-precision floating point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Fast Block LMS Filter   Signal Processing Blockset

Kalman Adaptive Filter   Signal Processing Blockset

Block LMS Filter

LMS Filter	Signal Processing Blockset
RLS Filter	Signal Processing Blockset

See "Adaptive Filters" on page 3-53 for related information.

Purpose   Buffer input sequence to smaller or larger frame size

Library   Signal Management / Buffers
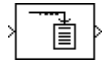
Description   The Buffer block redistributes the input samples to a new frame size,
                   larger or smaller than the input frame size. Buffering to a larger
                   frame size yields an output with a *slower* frame rate than the input, as
                   illustrated below for scalar input.

"fast-time" input
(frame size = 1, sample period = T_{si})

"slow-time" output
(frame size = 3, frame period = $3*T_{si}$)
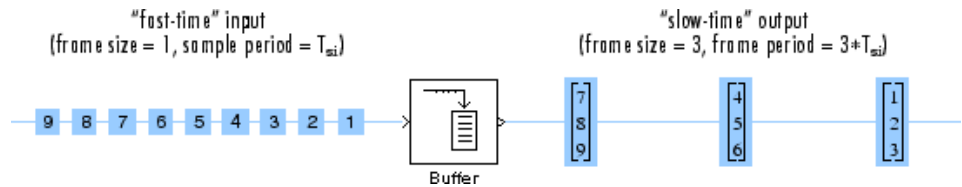
Buffering to a smaller frame size yields an output with a *faster* frame
rate than the input, as illustrated below for scalar output.

"slow-time" input
(frame size = 3, frame period = $3*T_{si}$)

"fast-time" output
(frame size = 1, sample period = T_{si})
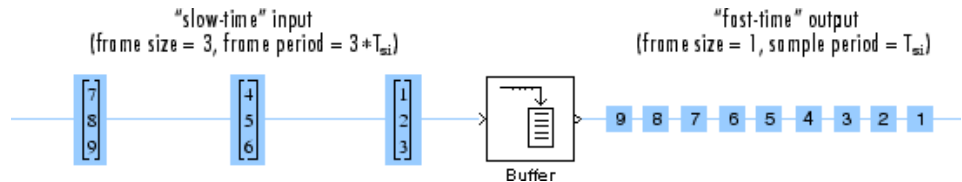
The block coordinates the output *frame size* and *frame rate* of
nonoverlapping buffers so that the sample period of the signal is the
same at both the input and output, $T_{so} = T_{si}$.

This block supports triggered subsystems when the block's input and
output rates are the same.

Sample-Based Operation

Sample-based inputs are interpreted by the Buffer block as independent
channels of data. Thus, a sample-based length-*N* vector input is
interpreted as *N* independent samples.

Buffer

In sample-based operation, the Buffer block creates frame-based outputs from sample-based inputs. A sequence of sample-based length-N vector inputs (1-D, 2-D row, or 2-D column) is buffered into an M_o-by-N matrix, where M_o is specified by the **Output buffer size** parameter ($M_o > 1$). That is, each input vector becomes a *row* in the N-channel frame-based output matrix. When $M_o = 1$, the input is simply passed through to the output, and retains the same dimension.

Sample-based full-dimension matrix inputs are not accepted.

The **Buffer overlap** parameter, L, specifies the number of samples (rows) from the current output to repeat in the next output, where $L < M_o$. For $0 \leq L < M_o$, the number of *new* input samples that the block acquires before propagating the buffered data to the output is the difference between the **Output buffer size** and **Buffer overlap**, $M_o - L$.

The output frame period is $(M_o - L) * T_{si}$, which is *equal* to the input sequence sample period, T_{si}, when the **Buffer overlap** is $M_o - 1$. For $L < 0$, the block simply discards L input samples after the buffer fills, and outputs the buffer with period $(M_o - L) * T_{si}$, which is longer than the zero-overlap case.

In the model below, the block buffers a four-channel sample-based input using a **Output buffer size** of 3 and a **Buffer overlap** of 1.
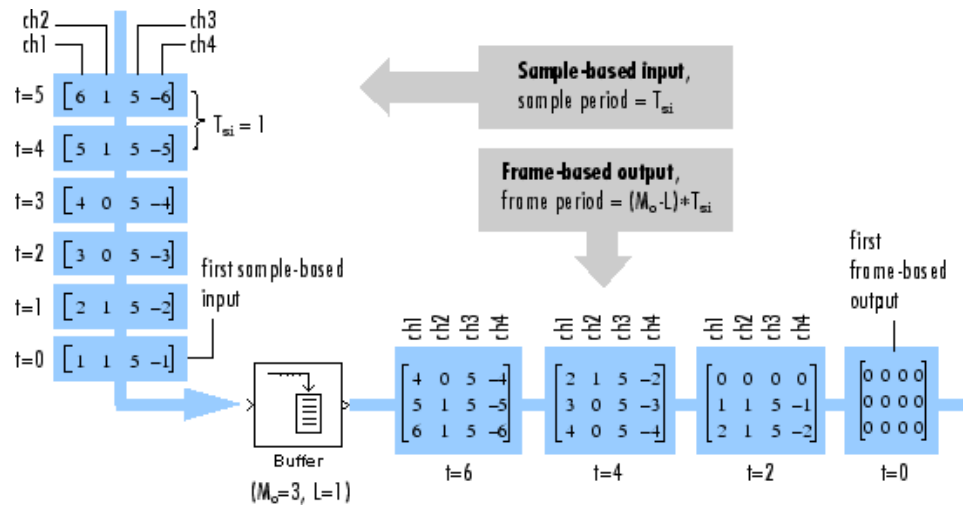
Notice that the input vectors do not begin appearing at the output until the second row of the second matrix. This is due to the block's latency. The first output matrix (all zeros in this example) reflects the block's **Initial conditions** setting, while the first row of zeros in the second output is a result of the one-sample overlap between consecutive output frames.

You can use the rebuffer_delay function with a frame size of 1 to precisely compute the delay (in samples) for sample-based signals. For the previous example,

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

This agrees with the four samples of delay (zeros) per channel shown in the previous figure.

Frame-Based Operation

In frame-based operation, the Buffer block redistributes the samples in the input frame to an output frame with a new size and rate. A

Buffer

sequence of M_i-by-N matrix inputs is buffered into a sequence of M_o-by-N frame-based matrix outputs, where M_o is the output frame size specified by the **Output buffer size** parameter (that is, the number of consecutive samples from the input frame to buffer into the output frame). M_o can be greater or less than the input frame size, M_i. Each of the N input channels is buffered independently.

The **Buffer overlap** parameter, L, specifies the number of samples (rows) from the current output to repeat in the next output, where $L <$ M_o. For $0 \leq L < M_o$, the number of *new* input samples the block acquires before propagating the buffered data to the output is the difference between the **Output buffer size** and **Buffer overlap**, M_o-L.

The input frame period is $M_i * T_{si}$, where T_{si} is the sample period. The output frame period is $(M_o$-$L) * T_{si}$, which is *equal* to the sequence sample period when the **Buffer overlap** is M_o-1. The output sample period is therefore related to the input sample period by

$$T_{so} = \frac{(M_o - L)T_{si}}{M_o}$$

Negative **Buffer overlap** values are not permitted.

In the model below, the block buffers a two-channel frame-based input using a **Output buffer size** of 3 and a **Buffer overlap** of 1.
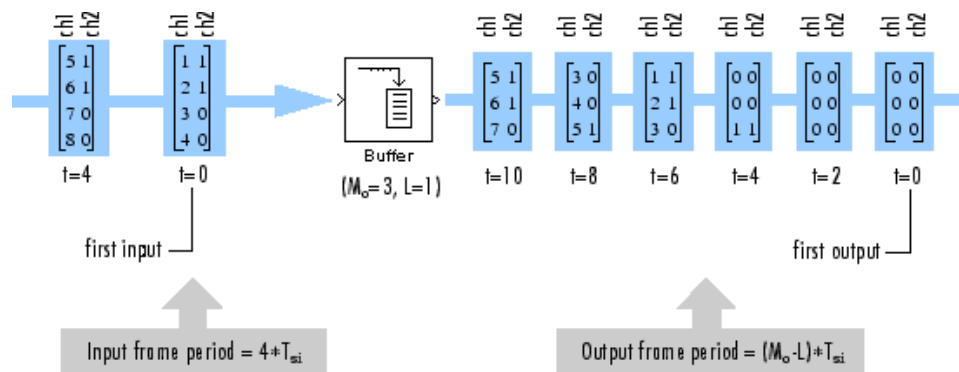
Notice that the sequence is delayed by eight samples, which is the latency of the block in the Simulink multitasking mode for the parameter settings of this example. The first eight output samples therefore adopt the value specified for the **Initial conditions**, which is assumed here to be zero. Use the rebuffer_delay function to determine the block's latency for any combination of frame size and overlap.

Zero Latency

In the Simulink single tasking mode, the Buffer block has *zero tasking latency* (the first input sample, received at $t=0$, appears as the first output sample) for the following special cases:

- Scalar input and output ($M_o = M_i = 1$) with zero or negative **Buffer overlap** ($L \leq 0$)

- Input frame size is integer multiple of the output frame size ($M_i = kM_o$, for k an integer) with zero **Buffer overlap** ($L = 0$); notable cases of this include

  - Any input frame size M_i with scalar output ($M_o = 1$) and zero **Buffer overlap** ($L = 0$)

  - Equal input and output frame sizes ($M_o = M_i$) with zero **Buffer overlap** ($L = 0$)

Nonzero Latency

- "Sample-Based Operation" on page 10-39
- "Frame-Based Operation" on page 10-40

Sample-Based Operation

For all cases of *sample-based single-tasking* operation other than those listed above, the Buffer block's buffer is initialized to the value(s) specified by the **Initial conditions** parameter, and the block reads from this buffer to generate the first D output samples, where

$$D = \begin{cases} M_o + L & (L \geq 0) \\ M_o & (L < 0) \end{cases}$$

When the **Buffer overlap**, L, is zero, the **Initial conditions** parameter can be a scalar to be repeated across the first M_o output samples, or a length-M_o vector containing the values of the first M_o output samples. For nonzero **Buffer overlap**, the **Initial conditions** parameter must be a scalar.

Frame-Based Operation

For *frame-based single-tasking* operation and all *multitasking* operation, use the rebuffer_delay function to compute the exact delay (in samples) that the Buffer block introduces for a given combination of buffer size and buffer overlap.

For general buffering between arbitrary frame sizes, the **Initial conditions** parameter must be a scalar value, which is then repeated across all elements of the initial output(s). However, in the special case where the *input* is 1-by-N (and the block's output is therefore an M_o-by-N matrix), **Initial conditions** can be

- An M_o-by-N matrix

- A length-M_o vector to be repeated across all columns of the initial output(s)

- A scalar to be repeated across all elements of the initial output(s)

In the special case where the *output* is 1-by-N (the result of unbuffering an M_i-by-N frame-based matrix), **Initial conditions** can be

- A vector containing M_i samples to output sequentially for each channel during the first M_i sample times

- A scalar to be repeated across all elements of the initial output(s)

Note For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" on page 2-56 and "Models with Multiple Sample Rates" in the Real-Time Workshop User's Guide documentation.

Dialog Box
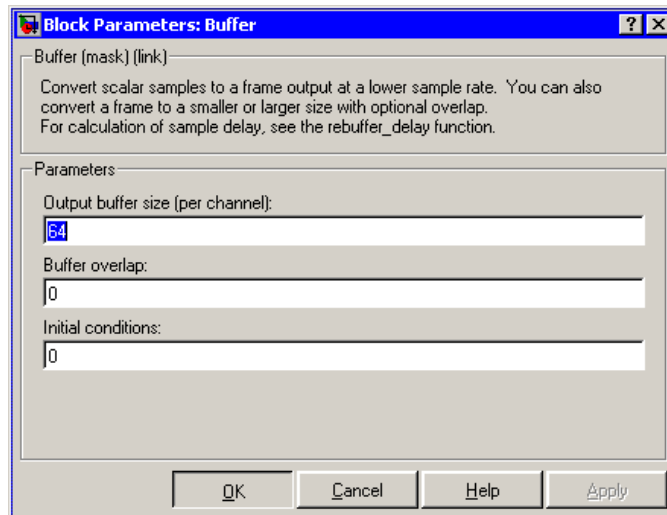
Output buffer size

The number of consecutive samples, M_o, from each channel to buffer into the output frame.

Buffer overlap

The number of samples, L, by which consecutive output frames overlap.

Initial conditions

The value of the block's initial output for cases of nonzero latency; a scalar, vector, or matrix.

Buffer

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • Custom data types • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • Custom data types • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Delay Line	Signal Processing Blockset
Unbuffer	Signal Processing Blockset
rebuffer_delay	Signal Processing Blockset

See "Converting Sample and Frame Rates" on page 2-12 and "Converting Frame Status" on page 2-33 for more information.

Purpose   Compute estimate of autoregressive (AR) model parameters using Burg method

Library   Estimation / Parametric Estimation

Description

Burg AR Estimator block image
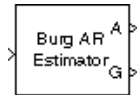
The Burg AR Estimator block uses the Burg method to fit an autoregressive (AR) model to the input data by minimizing (least squares) the forward and backward prediction errors while constraining the AR parameters to satisfy the Levinson-Durbin recursion.

The input is a sample-based vector (row, column, or 1-D) or frame-based vector (column only) representing a frame of consecutive time samples from a single-channel signal, which is assumed to be the output of an AR system driven by white noise. The block computes the normalized estimate of the AR system parameters, $A(z)$, independently for each successive input frame.

$$H(z) = \frac{G}{A(z)} = \frac{G}{1 + a(2)z^{-1} + \ldots + a(p+1)z^{-p}}$$

When you select the **Inherit estimation order from input dimensions** parameter, the order, p, of the all-pole model is one less that the length of the input vector. Otherwise, the order is the value specified by the **Estimation order** parameter.

The **Output(s)** parameter allows you to select between two realizations of the AR process:

- A — The top output, A, is a column vector of length p+1 with the same frame status as the input, and contains the normalized estimate of the AR model polynomial coefficients in descending powers of z.

    ```
    [1 a(2) ... a(p+1)]
    ```

- K — The top output, K, is a column vector of length p with the same frame status as the input, and contains the reflection coefficients (which are a secondary result of the Levinson recursion).

Burg AR Estimator

- `A` and `K` — The block outputs both realizations.

The scalar gain, G, is provided at the bottom output (`G`).

The following table compares the features of the Burg AR Estimator block to the Covariance AR Estimator, Modified Covariance AR Estimator, and Yule-Walker AR Estimator blocks.

	Burg AR Estimator	Covariance AR Estimator	Modified Covariance AR Estimator	Yule-Walker AR Estimator
`Characteristics`	Does not apply window to data	Does not apply window to data	Does not apply window to data	Applies window to data
	Minimizes the forward and backward prediction errors in the least squares sense, with the AR coefficients constrained to satisfy the L-D recursion	Minimizes the forward prediction error in the least squares sense	Minimizes the forward and backward prediction errors in the least squares sense	Minimizes the forward prediction error in the least squares sense (also called "autocorrelation method")
`Advantages`	Always produces a stable model			Always produces a stable model

	Burg AR Estimator	**Covariance AR Estimator**	**Modified Covariance AR Estimator**	**Yule-Walker AR Estimator**
`Disadvantages`		May produce unstable models	May produce unstable models	Performs relatively poorly for short data records
`Conditions for Nonsingularity`		Order must be less than or equal to half the input frame size	Order must be less than or equal to 2/3 the input frame size	Because of the biased estimate, the autocorrelation matrix is guaranteed to positive-definite, hence nonsingular

Dialog Box
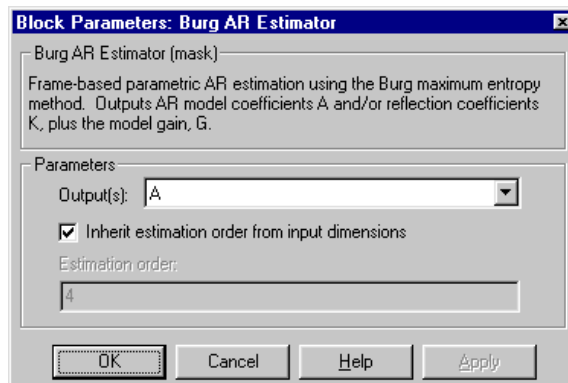
Output(s)

The realization to output, model coefficients, reflection coefficients, or both.

Burg AR Estimator

Inherit estimation order from input dimensions
> When selected, sets the estimation order p to one less than the length of the input vector.

Estimation order
> The order of the AR model, p. This parameter is enabled when you do not select **Inherit estimation order from input dimensions**.

References Kay, S. M. *Modern Spectral Estimation: Theory and Application.* Englewood Cliffs, NJ: Prentice-Hall, 1988.

Marple, S. L., Jr., *Digital Spectral Analysis with Applications.* Englewood Cliffs, NJ: Prentice-Hall, 1987.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
A	• Double-precision floating point • Single-precision floating point
G	• Double-precision floating point • Single-precision floating point

The output data type is the same as the input data type. To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Burg Method	Signal Processing Blockset
Covariance AR Estimator	Signal Processing Blockset
Modified Covariance AR Estimator	Signal Processing Blockset
Yule-Walker AR Estimator	Signal Processing Blockset
arburg	Signal Processing Toolbox

Burg Method

Purpose   Compute parametric spectral estimate using Burg method

Library   Estimation / Power Spectrum Estimation
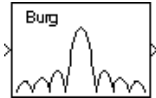
Description   The Burg Method block estimates the power spectral density (PSD)
of the input frame using the Burg method. This method fits an
autoregressive (AR) model to the signal by minimizing (least squares)
the forward and backward prediction errors while constraining the AR
parameters to satisfy the Levinson-Durbin recursion.

The input is a sample-based vector (row, column, or 1-D) or frame-based
vector (column only) representing a frame of consecutive time samples
from a single-channel signal. The block's output (a column vector) is
the estimate of the signal's power spectral density at N_{fft} equally spaced
frequency points in the range [0,F_s), where F_s is the signal's sample
frequency.

When you select the **Inherit estimation order from input
dimensions** parameter, the order of the all-pole model is one less that
the input frame size. Otherwise, the order is the value specified by the
Estimation order parameter. The spectrum is computed from the FFT
of the estimated AR model parameters.

When you select the **Inherit FFT length from estimation order**
parameter, N_{fft} is specified by the frame size of the input, which must
be a power of 2. When you do *not* select **Inherit FFT length from
estimation order**, N_{fft} is specified as a power of 2 by the **FFT length**
parameter, and the block zero pads or truncates the input to N_{fft} before
computing the FFT. The output is always sample based.

The Burg Method and Yule-Walker Method blocks return similar results
for large frame sizes. The following table compares the features of the
Burg Method block to the Covariance Method, Modified Covariance
Method, and Yule-Walker Method blocks.

	Burg	**Covariance**	**Modified Covariance**	**Yule-Walker**
`Characteristics`	Does not apply window to data	Does not apply window to data	Does not apply window to data	Applies window to data
	Minimizes the forward and backward prediction errors in the least squares sense, with the AR coefficients constrained to satisfy the L-D recursion	Minimizes the forward prediction error in the least squares sense	Minimizes the forward and backward prediction errors in the least squares sense	Minimizes the forward prediction error in the least squares sense (also called "autocorrelation method")
`Advantages`	High resolution for short data records	Better resolution than Y-W for short data records (more accurate estimates)	High resolution for short data records	Performs as well as other methods for large data records
	Always produces a stable model	Able to extract frequencies from data consisting of p or more pure sinusoids	Able to extract frequencies from data consisting of p or more pure sinusoids	Always produces a stable model
				Does not suffer spectral line-splitting

Burg Method

	Burg	Covariance	Modified Covariance	Yule-Walker
Disadvantages	Peak locations highly dependent on initial phase	May produce unstable models	May produce unstable models	Performs relatively poorly for short data records
	May suffer spectral line-splitting for sinusoids in noise, or when order is very large	Frequency bias for estimates of sinusoids in noise	Peak locations slightly dependent on initial phase	Frequency bias for estimates of sinusoids in noise
	Frequency bias for estimates of sinusoids in noise		Minor frequency bias for estimates of sinusoids in noise	
Conditions for Nonsingularity		Order must be less than or equal to half the input frame size	Order must be less than or equal to 2/3 the input frame size	Because of the biased estimate, the autocorrelation matrix is guaranteed to positive-definite, hence nonsingular

Examples   The dspsacomp demo compares the Burg method with several other spectral estimation methods.
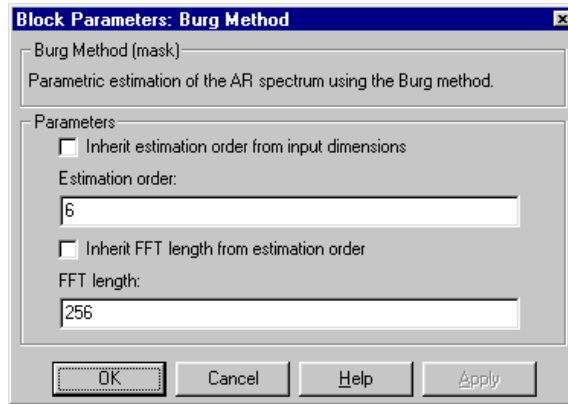
Dialog Box

Inherit estimation order from input dimensions

When selected, sets the estimation order to one less than the length of the input vector. Nontunable.

Estimation order

The order of the AR model. This parameter is enabled when you do not select **Inherit estimation order from input dimensions**. Nontunable.

Inherit FFT length from estimation order

When selected, uses the input frame size as the number of data points, N_{fft}, on which to perform the FFT. Nontunable.

FFT length

The number of data points, N_{fft}, on which to perform the FFT. When N_{fft} exceeds the input frame size, the frame is zero-padded as needed. This parameter is enabled when you do not select **Inherit FFT length from input dimensions**. Nontunable.

References   Kay, S. M. *Modern Spectral Estimation: Theory and Application.* Englewood Cliffs, NJ: Prentice-Hall, 1988.

Orfanidis, J. S. *Optimum Signal Processing: An Introduction.* 2nd ed. New York, NY: Macmillan, 1985.

Burg Method

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point

The output data type is the same as the input data type. To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Burg AR Estimator	Signal Processing Blockset
Covariance Method	Signal Processing Blockset
Modified Covariance Method	Signal Processing Blockset
Short-Time FFT	Signal Processing Blockset
Yule-Walker Method	Signal Processing Blockset
pburg	Signal Processing Toolbox

See "Power Spectrum Estimation" on page 6-6 for related information.

Purpose   Generate error when input signal does or does not match selected attributes exactly

Library   Signal Management / Signal Attributes

Description The Check Signal Attributes block terminates the simulation with an error when the input characteristics differ from those specified by the block parameters.

When the **Error when input** parameter is set to `Does not match attributes exactly`, the block generates an error only when the input possesses *none* of the attributes specified by the other parameters. Signals that possess *at least one* of the specified attributes are propagated to the output unaltered, and do not generate an error.

When the **Error when input** parameter is set to `Matches attributes exactly`, the block generates an error only when the input possesses *all* attributes specified by the other parameters. Signals that do not possess *all* of the specified attributes are propagated to the output unaltered, and do not generate an error.

Signal Attributes

The Check Signal Attributes block can test for up to five different signal attributes, as specified by the following parameters. When you select the `Ignore` in any parameter, the block does not check the signal for the corresponding attribute. For example, when **Complexity** is set to `Ignore`, neither real nor complex inputs cause the block to generate an error. The attributes are

- **Complexity**

  Checks whether the signal is real or complex. (Note that this information can be displayed in a model by attaching a Probe block with **Probe complex signal** selected. Alternatively, in the model window, from the **Format** menu, point to **Port/Signal Displays**, and select **Port Data Types**.)

10-53

Check Signal Attributes

- **Frame status**

  Checks whether the signal is frame based or sample based. (Note that Simulink displays sample-based signals using a single line, , and frame-based signals using a double line,

- **Dimensionality**

  Checks the dimension of signal for compliance (Is...) or noncompliance (Is not...) with the attributes in the subordinate **Dimension** menu, which are shown in the table below. *M* and *N* are positive integers unless otherwise indicated below.

Dimensions	Is...	Is not...
1-D	1-D vector, 1-D scalar	*M*-by-*N* matrix, 1-by-*N* matrix (row vector), *M*-by-1 matrix (column vector), 1-by-1 matrix (2-D scalar)
2-D	*M*-by-*N* matrix, 1-by-*N* matrix (row vector), *M*-by-1 matrix (column vector), 1-by-1 matrix (2-D scalar)	1-D vector, 1-D scalar
Scalar (1-D or 2-D)	1-D scalar, 1-by-1 matrix (2-D scalar)	1-D vector with length>1, *M*-by-*N* matrix with *M*>1 and/or *N*>1

Dimensions	Is...	Is not...
Vector (1-D or 2-D)	1-D vector, 1-D scalar, 1-by-N matrix (row vector), M-by-1 matrix (column vector), 1-by-1 matrix (2-D scalar) Vector (1-D or 2-D) or scalar	M-by-N matrix with $M>1$ and $N>1$
Row Vector (2-D)	1-by-N matrix (row vector), 1-by-1 matrix (2-D scalar) Row vector (2-D) or scalar	1-D vector, 1-D scalar, M-by-N matrix with $M>1$
Column Vector (2-D)	M-by-1 matrix (column vector), 1-by-1 matrix (2-D scalar) Column vector (2-D) or scalar	1-D vector, 1-D scalar, M-by-N matrix with $N>1$
Full matrix	M-by-N matrix with $M>1$ and $N>1$	1-D vector, 1-D scalar, 1-by-N matrix (row vector), M-by-1 matrix (column vector), 1-by-1 matrix (2-D scalar)
Square matrix	M-by-N matrix with $M=N$, 1-D scalar,	M-by-N matrix with $M \neq N$, 1-D vector,

Dimensions	Is...	Is not...
	1-by-1 matrix (2-D scalar	1-by-*N* matrix (row vector), *M*-by-1 matrix (column vector)

If, in the model window, from the **Format** menu, you point to **Port/Signal Displays**, and select **Signal Dimensions**, Simulink displays the size of a 1-D vector signal as an unbracketed integer, and displays the dimension of a 2-D signal as a pair of bracketed integers, [MxN]. Simulink *does not display* any size information for a 1-D or 2-D scalar signal. Dimension information for a signal can also be displayed in a model by attaching a Probe block with **Probe signal dimensions** selected.

- **Data type**

  Checks the signal data type for compliance (Is...) or noncompliance (Is not...) with the attributes in the subordinate **General data type** menu, which are shown in the table below. Any of the specific data types listed in the Is... column below can be individually selected from the subordinate **Specific data type** menu.

General Data Type	Is...	Is not...
Boolean	boolean	single, double, uint8, int8, uint16, int16, uint32, int32, fixed-point
Floating-point	single, double	boolean, uint8, int8, uint16, int16, uint32, int32, fixed-point

General Data Type	Is...	Is not...
Fixed-point	fixed-point	boolean, uint8, int8, uint16, int16, uint32, int32, single, double
Integer	Signed integer int8, int16, int32 Unsigned integer uint8, uint16, uint32	boolean, single, double

To display data type information, in your model window, from the **Format** menu, point to **Port/Signal Displays**, and select **Port Data Types**.

- **Sample mode**

  Checks whether the signal is discrete-time or continuous-time. (If, from the **Format** menu, you point to **Port/Signal Displays**, and select **Sample Time Colors**, Simulink displays continuous-time signal lines in black or grey and discrete-time signal lines in colors corresponding to the relative rate. When a Probe block with **Probe sample time** enabled is attached to a continuous-time signal, the block icon displays the string Ts:[0 x], where x is the sample time offset. When a Probe block is attached to a discrete-time signal, the block icon displays the string Ts:[t 0] for a sample-based signal or Tf:[t 0] for a frame-based signal, where t is the nonzero sample period or frame period, respectively. Frame-based signals are almost always discrete time.

Check Signal Attributes

Dialog Box
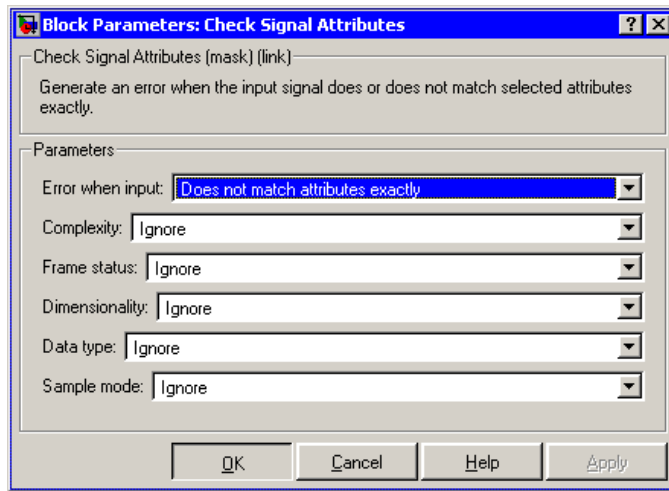
Error when input

Specifies whether the block generates an error when the input possesses *none* of the required attributes (Does not match attributes exactly), or when the input possesses *all* of the required attributes (Matches attributes exactly).

Complexity

The complexity for which the input should be checked, Real or Complex. When you select Ignore from the list, the block does not check the input's complexity.

Frame status

The frame status for which the input should be checked, Sample-based or Frame-based. When you select Ignore from the list, the block does not check the input's frame status.

Dimensionality

Specifies whether the input should be checked for compliance (Is...) or noncompliance (Is not...) with the attributes in the subordinate **Dimension** menu. When you select Ignore from the list, the block does not check the input's dimensionality.

Data type

Specifies whether the input should be checked for compliance (Is...) or noncompliance (Is not...) with the attributes in the subordinate **General data type** menu. When you select Ignore from the list, the block does not check the input's data type.

Sample mode

The sample mode for which the input should be checked, Discrete or Continuous. When you select Ignore from the list, the block does not check the input's sample mode.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • Custom data types • Boolean • 8, 16, and 32-bit signed integers • 8, 16, and 32-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • Custom data types • Boolean • 8, 16, and 32-bit signed integers • 8, 16, and 32-bit unsigned integers

Check Signal Attributes

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Buffer	Signal Processing Blockset
Convert 1-D to 2-D	Signal Processing Blockset
Convert 2-D to 1-D	Signal Processing Blockset
Data Type Conversion	Simulink
Frame Status Conversion	Signal Processing Blockset
Inherit Complexity	Signal Processing Blockset
Probe	Simulink
Reshape	Simulink
Submatrix	Signal Processing Blockset

Purpose   Generate swept-frequency cosine (chirp) signal

Library   Signal Processing Sources
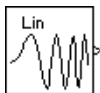
Description   The Chirp block outputs a swept-frequency cosine (chirp) signal with unity amplitude and continuous phase. To specify the desired output chirp signal, you must define its instantaneous frequency function, also known as the output frequency sweep. The frequency sweep can be linear, quadratic, or logarithmic, and repeats once every **Sweep time** by default. See other sections of this reference page for more details about the block.

Sections of This Reference Page

Variables Used in This Reference Page

f_0	**Initial frequency** parameter (Hz)
$f_i(t_g)$	**Target frequency** parameter (Hz)

t_g	**Target time** parameter (seconds)
T_{sw}	**Sweep time** parameter (seconds)
ϕ_0	**Initial phase** parameter (radians)
$\psi(t)$	Phase of the chirp signal (radians)
$f_i(t)$	User-specified output instantaneous frequency function (Hz); user-specified sweep
$f_{i(actual)}(t)$	Actual output instantaneous frequency function (Hz); actual output sweep
$y_{chirp}(t)$	Output chirp function

Setting the Output Frame Status

Use **Samples per frame** parameter to set the block's output frame status, as summarized in the following table. The **Sample time** parameter sets the sample time of both sample- and frame-based outputs.

Setting of Samples Per Frame Parameter	Output Frame Status
1	Sample based
n (any integer greater than 1)	Frame based, frame size n

Shaping the Frequency Sweep by Setting Frequency Sweep and Sweep Mode

The basic shape of the output instantaneous frequency sweep, $f_i(t)$, is set by the **Frequency sweep** and **Sweep mode** parameters, described in the following table.

Parameters for Setting Sweep Shape	Possible Settings	Parameter Description
Frequency sweep	Linear Quadratic Logarithmic Swept cosine	Determines whether the sweep frequencies vary linearly, quadratically, or logarithmically. Linear and swept cosine sweeps both vary linearly.
Sweep mode	Unidirectional Bidirectional	Determines whether the sweep is unidirectional or bidirectional. For details, see "Unidirectional and Bidirectional Sweep Modes" on page 10-64.

The following diagram illustrates the possible shapes of the frequency sweep that you can obtain by setting the **Frequency sweep** and **Sweep mode** parameters.

Chirp

Possible Shapes of the Output Instantaneous Frequency Sweep
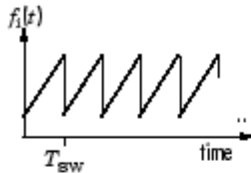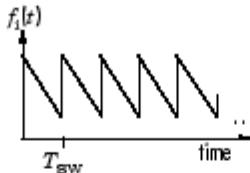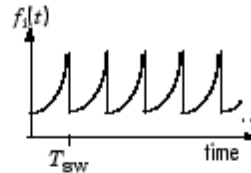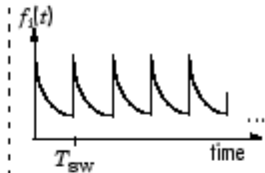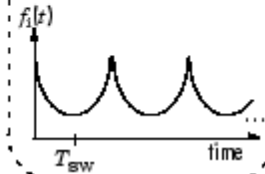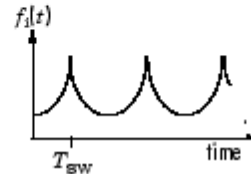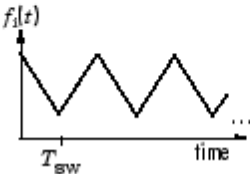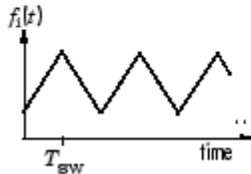
For information on how to set the frequency values in your sweep, see "Setting Instantaneous Frequency Sweep Values" on page 10-65.

Unidirectional and Bidirectional Sweep Modes

The **Sweep mode** parameter determines whether your sweep is unidirectional or bidirectional, which affects the shape of your output frequency sweep (see "Shaping the Frequency Sweep by Setting Frequency Sweep and Sweep Mode" on page 10-62). The following table describes the characteristics of unidirectional and bidirectional sweeps.

Sweep Mode Parameter Settings	Sweep Characteristics
Unidirectional	• Lasts for one **Sweep time**, T_{sw} • Repeats once every T_{sw}
Bidirectional	• Lasts for twice the **Sweep time**, $2*T_{\mathrm{sw}}$ • Repeats once every $2*T_{\mathrm{sw}}$ • First half is identical to its unidirectional counterpart. • Second half is a mirror image of the first half.

The following diagram illustrates a linear sweep in both sweep modes. For information on setting the frequency values in your sweep, see "Setting Instantaneous Frequency Sweep Values" on page 10-65.

Unidirectional Linear Sweep
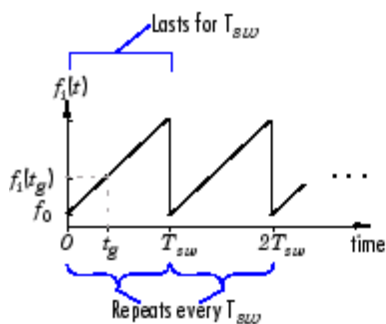
Bidirectional Linear Sweep
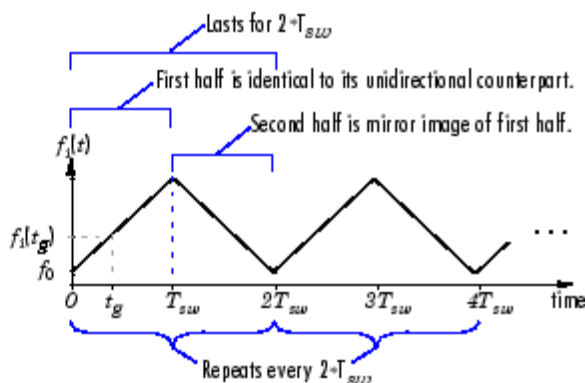
Setting Instantaneous Frequency Sweep Values

Set the following parameters to tune the frequency values of your output frequency sweep. Note that because this is a source block, the

simulation will pause while the block dialog box is open. You must close
the dialog box by clicking **OK** to resume the simulation.

- **Initial frequency** (Hz), f_0

- **Target frequency** (Hz), $f_i(t_g)$

- **Target time** (seconds), t_g

The following table summarizes the sweep values at specific times
for all **Frequency sweep** settings. For information on the formulas
used to compute sweep values at other times, see "Block Computation
Methods" on page 10-66.

Instantaneous Frequency Sweep Values

Frequency Sweep	Sweep Value at $t = 0$	Sweep Value at $t = t_g$	Time when Sweep Value Is Target Frequency, $f_i(t_g)$
Linear	f_0	$f_i(t_g)$	t_g
Quadratic	f_0	$f_i(t_g)$	t_g
Logarithmic	f_0	$f_i(t_g)$	t_g
Swept cosine	f_0	$2f_i(t_g) - f_0$	$t_g/2$

Block Computation Methods

The Chirp block uses one of two formulas to compute the block output,
depending on the **Frequency Sweep** parameter setting. For details,
see the following sections:

- "Equations for Output Computation" on page 10-67

- "Output Computation Method for Linear, Quadratic, and Logarithmic
  Frequency Sweeps" on page 10-68

- "Output Computation Method for Swept Cosine Frequency Sweep"
  on page 10-69

Equations for Output Computation

The following table shows the equations used by the block to compute the user-specified output frequency sweep, $f_i(t)$, the block output, $y_{chirp}(t)$, and the actual output frequency sweep, $f_{i(actual)}(t)$. The only time the user-specified sweep is not the actual output sweep is when the **Frequency sweep** parameter is set to Swept cosine.

Note The following equations apply only to unidirectional sweeps in which $f_i(0) < f_i(tg)$. To derive equations for other cases, you might find it helpful to examine the following table and the diagram in "Shaping the Frequency Sweep by Setting Frequency Sweep and Sweep Mode" on page 10-62.

The table below contains the following variables:

- $f_i(t)$ — the user-specified frequency sweep

- $f_{i(actual)}(t)$ — the actual output frequency sweep, usually equal to $f_i(t)$

- $y(t)$ — the Chirp block output

- $\psi(t)$ — the phase of the chirp signal, where $\psi(0) = 0$, and $2\pi f_i(t)$ is the derivative of the phase

$$f_i(t) = \frac{1}{2\pi} \cdot \frac{d\psi(t)}{dt}$$

- ϕ_0 — the **Initial phase** parameter value, where $y_{chirp}(0) = \cos(\phi_0)$

Chirp

Equations Used by the Chirp Block for Unidirectional Positive Sweeps

Frequency Sweep	Block Output Chirp Signal	User-Specified Frequency Sweep, $f_i(t)$	β	Actual Frequency Sweep, $f_{i(actual)}(t)$
Linear	$y(t) = \cos(\psi(t) + \phi_0)$	$f_i(t) = f_0 + \beta t$	$\beta = \dfrac{f_i(t_g) - f_0}{t_g}$	$f_{i(actual)}(t) = f_i(t)$
Quadratic	Same as Linear	$f_i(t) = f_0 + \beta t^2$	$\beta = \dfrac{f_i(t_g) - f_0}{t_g^2}$	$f_{i(actual)}(t) = f_i(t)$
Logarithmic	Same as Linear	$F_i(t) = f_0 \left(\dfrac{f_i(t_g)}{f_0} \right)^{\frac{t}{t_g}}$ Where $f_i(t_g) > f_0 > 0$	N/A	$f_{i(actual)}(t) = f_i(t)$
Swept cosine	$y(t) = \cos(2\pi f_i(t)t + \phi_0)$	Same as Linear	Same as Linear	$f_{i(actual)}(t) = f_i(t) + \beta t$

Output Computation Method for Linear, Quadratic, and Logarithmic Frequency Sweeps

The derivative of the phase of a chirp function gives the instantaneous frequency of the chirp function. The Chirp block uses this principle to calculate the chirp output when the **Frequency Sweep** parameter is set to Linear, Quadratic, or Logarithmic.

$$y_{chirp}(t) = \cos(\psi(t) + \phi_0)$$ Linear, quadratic, or logarithmic chirp signal with phase $\psi(t)$

$$f_i(t) = \frac{1}{2\pi} \cdot \frac{d\psi(t)}{dt}$$ Phase derivative is instantaneous frequency

For instance, if you want a chirp signal with a linear instantaneous frequency sweep, you should set the **Frequency Sweep** parameter to Linear, and tune the linear sweep values by setting other parameters

appropriately. Note that because this is a source block, the simulation will pause while the block dialog box is open. You must close the dialog box by clicking **OK** to resume the simulation. The block will output a chirp signal, the phase derivative of which is the specified linear sweep. This ensures that the instantaneous frequency of the output is the linear sweep you desired. For equations describing the linear, quadratic, and logarithmic sweeps, see "Equations for Output Computation" on page 10-67.

Output Computation Method for Swept Cosine Frequency Sweep

To generate the swept cosine chirp signal, the block sets the swept cosine chirp output as follows.

$$y_{chirp}(t) = \cos(\psi(t) + \phi_0) = \cos(2\pi f_i(t)t + \phi_0)$$

Swept cosine chirp output (instantaneous frequency equation, shown above, does not hold.)

Note that the instantaneous frequency equation, shown above, does not hold for the swept cosine chirp, so the user-defined frequency sweep, $f_i(t)$, is not the actual output frequency sweep, $f_{i(\text{actual})}(t)$, of the swept cosine chirp. Thus, the swept cosine output might not behave as you expect. To learn more about swept cosine chirp behavior, see "Cautions Regarding the Swept Cosine Sweep" on page 10-69 and "Equations for Output Computation" on page 10-67.

Cautions Regarding the Swept Cosine Sweep

When you want a linearly swept chirp signal, we recommend you use a linear frequency sweep. Though a swept cosine frequency sweep also yields a linearly swept chirp signal, the output might have unexpected frequency content. For details, see the following two sections.

Chirp

Swept Cosine Instantaneous Output Frequency at the Target Time is not the Target Frequency

The swept cosine sweep value at the **Target time** is not necessarily the **Target frequency**. This is because the user-specified sweep is not the actual frequency sweep of the swept cosine output, as noted in "Output Computation Method for Swept Cosine Frequency Sweep" on page 10-69. See the table Instantaneous Frequency Sweep Values on page 10-66 for the actual value of the swept cosine sweep at the **Target time**.

Swept Cosine Output Frequency Content May Greatly Exceed Frequencies in the Sweep

In **Swept cosine** mode, you should not set the parameters so that $1/T_{sw}$ is very large compared to the values of the **Initial frequency** and **Target frequency** parameters. In such cases, the actual frequency content of the swept cosine sweep might be closer to $1/T_{sw}$, far exceeding the **Initial frequency** and **Target frequency** parameter values.

Dialog Box
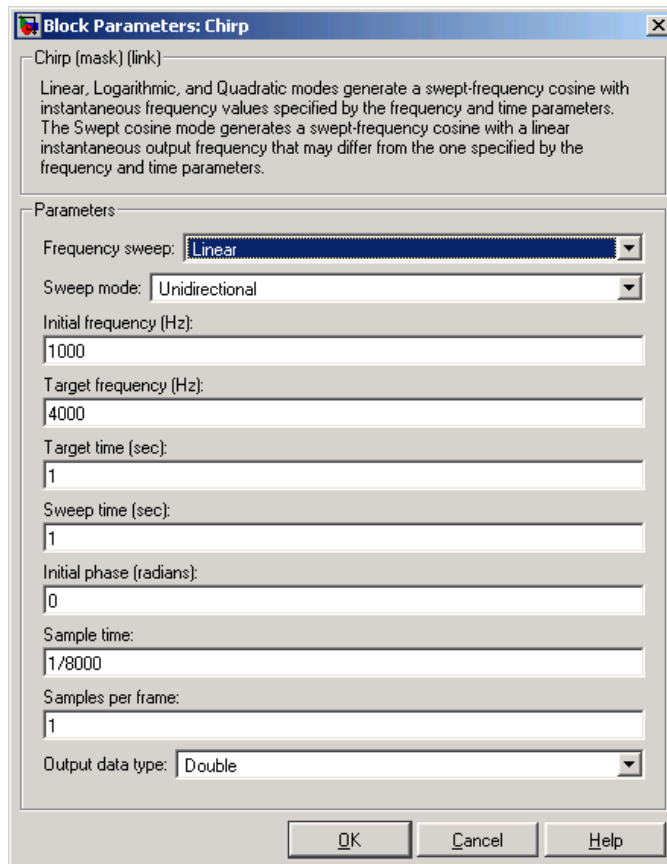
Opening this dialog box causes a running simulation to pause. See "Changing Source Block Parameters" in the online Simulink documentation for details.

Frequency sweep

The type of output instantaneous frequency sweep, $f_i(t)$: Linear, Logarithmic, Quadratic, or Swept cosine. Tunable.

Chirp

Sweep mode

> The directionality of the chirp signal: `Unidirectional` or `Bidirectional`. Nontunable.

Initial frequency (Hz)

> For `Linear`, `Quadratic`, and `Swept cosine` sweeps, the initial frequency, f_0, of the output chirp signal. For `Logarithmic` sweeps, **Initial frequency** is one less than the actual initial frequency of the sweep. Also, when the sweep is `Logarithmic`, you must set the **Initial frequency** to be less than the **Target frequency**. Tunable.

Target frequency (Hz)

> For `Linear`, `Quadratic`, and `Logarithmic` sweeps, the instantaneous frequency, $f_i(t_g)$, of the output at the **Target time**, t_g. For a `Swept cosine` sweep, **Target frequency** is the instantaneous frequency of the output at half the **Target time**, $t_g/2$. When **Frequency sweep** is `Logarithmic`, you must set the **Target frequency** to be greater than the **Initial frequency** Tunable.

Target time (sec)

> For `Linear`, `Quadratic`, and `Logarithmic` sweeps, the time, t_g, at which the **Target frequency**, $f_i(t_g)$, is reached by the sweep. For a `Swept cosine` sweep, **Target time** is the time at which the sweep reaches $2f_i(t_g) - f_0$. You must set **Target time** to be *no greater than* **Sweep time,** $T_{sw} \geq t_g$. Tunable.

Sweep time (sec)

> In `Unidirectional` **Sweep mode**, the **Sweep time**, Tsw, is the period of the output frequency sweep. In `Bidirectional` **Sweep mode**, the **Sweep time** is half the period of the output frequency sweep. You must set **Sweep time** to be no less than **Target time**, $T_{sw} \geq t_g$. Tunable.

Initial phase (radians)

> The phase, ϕ_0, of the cosine output at $t=0$; $y_{chirp}(t) = \cos(\phi_0)$. Tunable.

Sample time

The sample period, T_s, of the output. The output frame period is $M_o * T_s$.

Samples per frame

The number of samples, M_o, to buffer into each output frame.

Output data type

The data type of the output, single-precision or double-precision.

Examples

The first few examples demonstrate how to use the Chirp block's main parameters, how to view the output in the time domain, and how to view the output spectrogram:

- "Example 1: Setting a Final Frequency Value for Unidirectional Sweeps" on page 10-73
- "Example 2: Bidirectional Sweeps" on page 10-75
- "Example 3: When Sweep Time is Greater Than Target Time" on page 10-76

Examples 4 and 5 illustrate Chirp block settings that might produce unexpected outputs:

- "Example 4: Output Sweep with Negative Frequencies" on page 10-77
- "Example 5: Output Sweep with Frequencies Greater Than Half the Sampling Frequency" on page 10-78

Example 1: Setting a Final Frequency Value for Unidirectional Sweeps

Often times, you might want a unidirectional sweep for which you know the initial and final frequency values. You can specify the final frequency of a unidirectional sweep by setting **Target time** equal to **Sweep time**, in which case the **Target frequency** becomes the final frequency in the sweep. The following model demonstrates this method.

Chirp

This technique might not work for swept cosine sweeps. For details, see "Cautions Regarding the Swept Cosine Sweep" on page 10-69.

Open the Example 1 model by typing `doc_chirp_ref` at the MATLAB command line. You can also rebuild the model yourself; see the following list for model parameter settings (leave unlisted parameters in their default states).
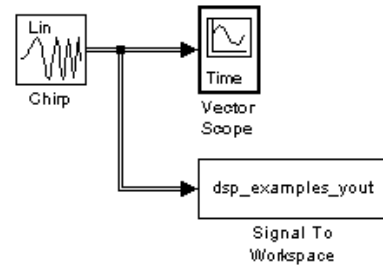
Since **Target time** is set to equal **Sweep time** (1 second), the **Target frequency** (25 Hz) is the final frequency of the unidirectional sweep.
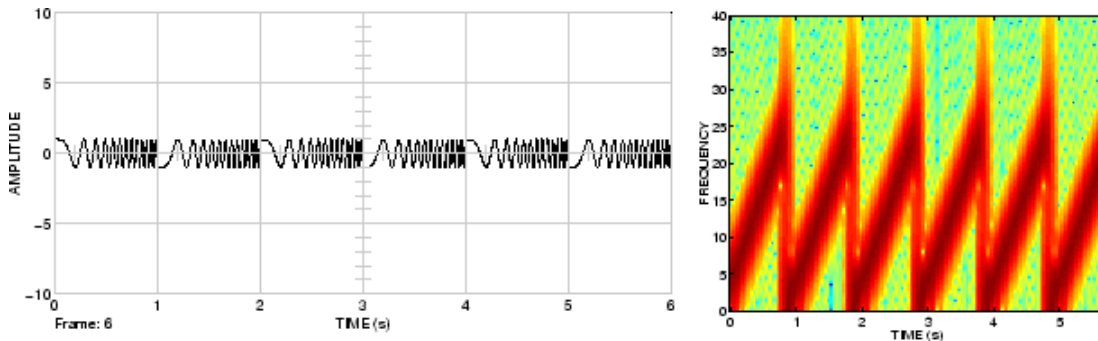
Run your model to see the time domain output, and then type the following command to view the chirp output spectrogram.

```
spectrogram(dsp_examples_yout,hamming(128),110,[0:.01:40],400)
```

Chirp Block Parameters for Example 1	
Frequency sweep	`Linear`
Sweep mode	`Unidirectional`
Initial frequency	0
Target frequency	25
Target time	1
Sweep time	1
Initial phase	0
Sample time	1/400
Samples per frame	400
Vector Scope Block Parameters for Example 1	
Input domain	`Time`
Time display span	6
Signal To Workspace Block Parameters for Example 1	
Variable name	`dsp_examples_yout`
Configuration Dialog Parameters for Example 1	
Stop time	5

Example 2: Bidirectional Sweeps

Change the **Sweep mode** parameter in the Example 1 model to
`Bidirectional`, and leave all other parameters the same to view the
following bidirectional chirp. Note that in the bidirectional sweep, the
period of the sweep is twice the **Sweep time** (2 seconds), whereas it was
one **Sweep time** (1 second) for the unidirectional sweep in Example 1.

Open the Example 2 model by typing `doc_chirp_ref2` at the MATLAB
command line.
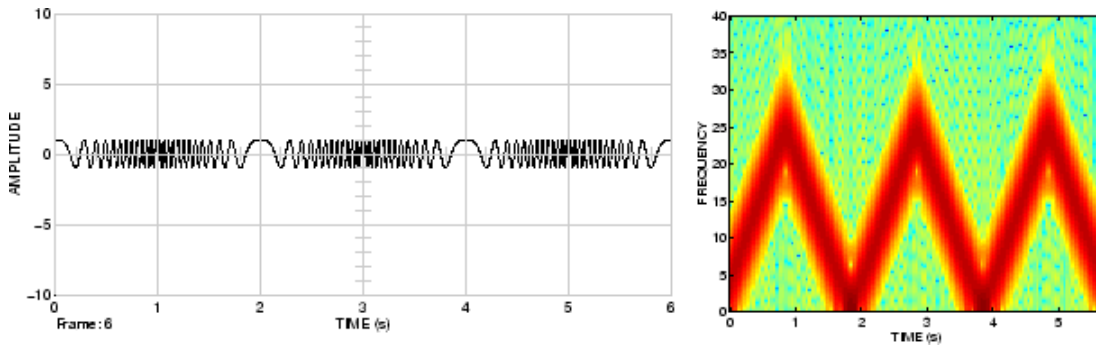
Run your model to see the time domain output, and then type the following command to view the chirp output spectrogram.

```
spectrogram(dsp_examples_yout,hamming(128),110,[0:.01:40],400)
```

Example 3: When Sweep Time is Greater Than Target Time

Setting **Sweep time** to 1.5 and leaving the rest of the parameters as in the Example 1 model gives the following output. The sweep still reaches the **Target frequency** (25 Hz) at the **Target time** (1 second), but since **Sweep time** is greater than **Target time**, the sweep continues on its linear path until one **Sweep time** (1.5 seconds) is traversed.

Unexpected behavior might arise when you set **Sweep time** greater than **Target time**; see "Example 4: Output Sweep with Negative Frequencies" on page 10-77 for details.

Open the Example 3 model by typing doc_chirp_ref3 at the MATLAB command line.

Run your model to see the time domain output, and then type the following command to view the chirp output spectrogram.

```
spectrogram(dsp_examples_yout,hamming(128),110,[0:.01:40],400)
```

Example 4: Output Sweep with Negative Frequencies

Modify the Example 1 model by changing **Sweep time** to 1.5, **Initial frequency** to 25, and **Target frequency** to 0. *The output chirp of this example might not behave as you expect* because the sweep contains negative frequencies between 1 and 1.5 seconds. The sweep reaches the **Target frequency** of 0 Hz at one second, then continues on its negative slope, taking on negative frequency values until it traverses one **Sweep time** (1.5 seconds).

The spectrogram might reflect negative sweep frequencies along the *x*-axis so they appear to be positive. If you unexpectedly get a chirp output with a spectrogram resembling the one below, your chirp's sweep might contain negative frequencies. See the next example for another possible unexpected chirp output.

Open the Example 4 model by typing doc_chirp_ref4 at the MATLAB command line.
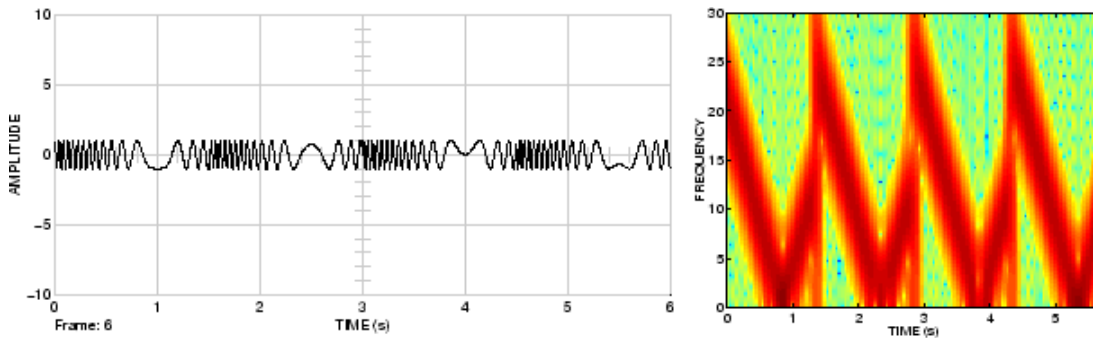
Run your model to see the time domain output, and then type the following command to view the chirp output spectrogram.
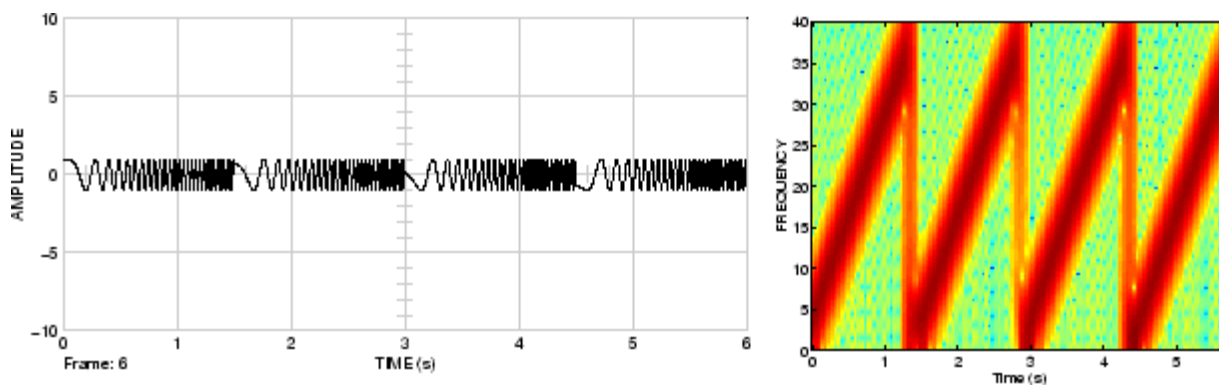
```
spectrogram(dsp_examples_yout,hamming(128),110,[0:.1:30],400);
```

Example 5: Output Sweep with Frequencies Greater Than Half the Sampling Frequency

Modify the Example 1 model by changing the **Target frequency** parameter to 275. *The output chirp of this model might not behave as you expect* because the sweep contains frequencies greater than half the sampling frequency (200 Hz), which causes aliasing. If you unexpectedly get a chirp output with a spectrogram resembling the one following, your chirp's sweep might contain frequencies greater than half the sampling frequency. See the previous example for another possible unexpected chirp output.

Open the Example 5 model by typing doc_chirp_ref5 at the MATLAB command line.
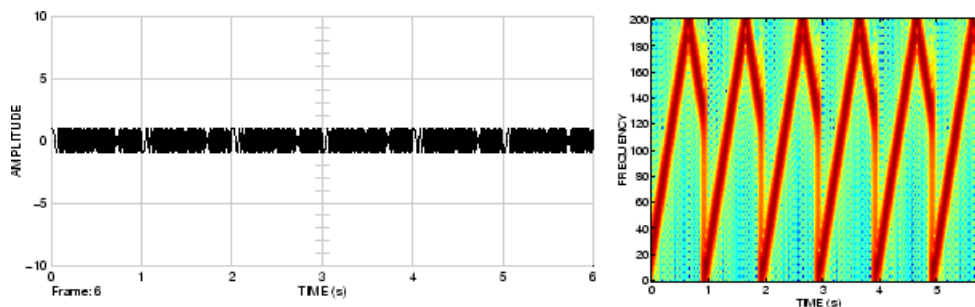
Run your model to see the time domain output, and then type the following command to view the chirp output spectrogram.

```
spectrogram(dsp_examples_yout,hamming(64),60,256,400)
```

**Supported Data Types**

- Double-precision floating point
- Single-precision floating point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Signal From Workspace	Signal Processing Blockset
Signal Generator	Simulink
Sine Wave	Signal Processing Blockset
chirp	Signal Processing Toolbox
spectrogram	Signal Processing Toolbox

Cholesky Factorization

Purpose   Factor square Hermitian positive definite matrix into triangular components.

Library   Math Functions / Matrices and Linear Algebra / Matrix Factorizations

Description   The Cholesky Factorization block uniquely factors the square Hermitian positive definite input matrix S as

$$S = LL^*$$
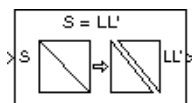
where L is a lower triangular square matrix with positive diagonal elements and L^* is the Hermitian (complex conjugate) transpose of L. The block outputs a matrix with lower triangle elements from L and upper triangle elements from L^*. The output is always sample based. The output is not in the same form as the output of the MATLAB chol function. In order to convert the output of the Cholesky Factorization block to the MATLAB form, use the following equation:

```
R = triu(LL');
```

Here, LL' is the output of the Cholesky Factorization block. Due to roundoff error, these equations do not produce a result that is exactly the same as the MATLAB result.

Sample-based
output with lower
triangle elements
from L and upper
triangle elements
from L*.

$$L = \begin{bmatrix} 3.00 & 0 & 0 \\ -0.33 & 2.81 & 0 \\ 0.67 & -1.70 & 1.91 \end{bmatrix}$$
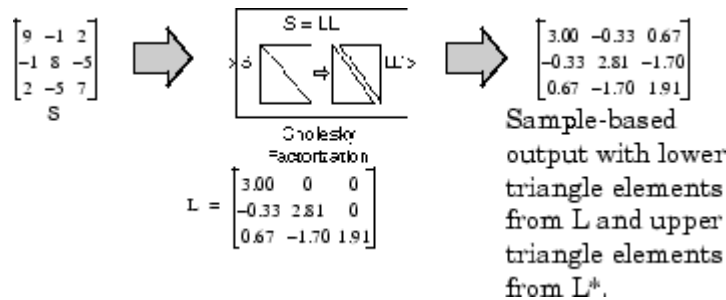
Block Output Composed of L and L*

Input Requirements for Valid Output

The block output is valid only when its input has the following characteristics:

- Hermitian — The block does *not* check whether the input is Hermitian; it uses only the diagonal and upper triangle of the input to compute the output.

- Real-valued diagonal entries — The block disregards any imaginary component of the input's diagonal entries.

- Positive definite — Set the block to notify you when the input is not positive definite as described in "Response to Nonpositive Definite Input" on page 10-81

Response to Nonpositive Definite Input

To generate a valid output, the block algorithm requires a positive definite input (see "Input Requirements for Valid Output" on page 10-81). Set the **Non-positive definite input** parameter to determine how the block responds to a nonpositive definite input:

- Ignore — Proceed with the computation and *do not* issue an alert. The output is *not* a valid factorization. A partial factorization will be present in the upper left corner of the output.

- Warning — Display a warning message in the MATLAB Command Window, and continue the simulation. The output is *not* a valid

Cholesky Factorization

factorization. A partial factorization will be present in the upper left corner of the output.

- Error — Display an error dialog and terminate the simulation.

Note The **Non-positive definite input** parameter is a diagnostic parameter. Like all diagnostic parameters on the Configuration Parameters dialog box, it is set to Ignore in the Real-Time Workshop code generated for this block.

Performance Comparisons with Other Blocks

Note that L and L^* share the same diagonal in the output matrix. Cholesky factorization requires half the computation of Gaussian elimination (LU decomposition), and is always stable.

Dialog Box
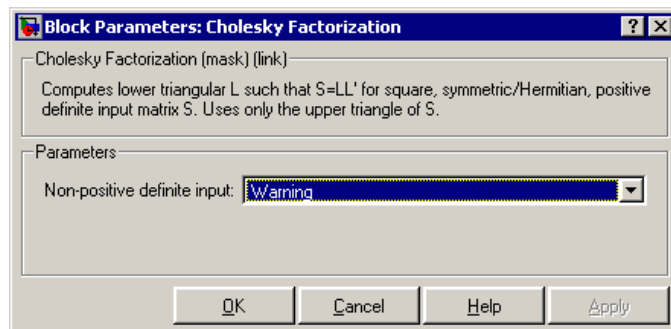
Non-positive definite input
Response to nonpositive definite matrix inputs: Ignore, Warning, or Error. See "Response to Nonpositive Definite Input" on page 10-81. Nontunable.

References Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

Supported Data Types

Port	Supported Data Types
S	• Double-precision floating point
	• Single-precision floating point
LL'	• Double-precision floating point
	• Single-precision floating point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Autocorrelation LPC	Signal Processing Blockset
Cholesky Inverse	Signal Processing Blockset
Cholesky Solver	Signal Processing Blockset
LDL Factorization	Signal Processing Blockset
LU Factorization	Signal Processing Blockset
QR Factorization	Signal Processing Blockset
chol	MATLAB

See "Factoring Matrices" on page 6-9 for related information.

Cholesky Inverse

Purpose   Compute inverse of Hermitian positive definite matrix using Cholesky factorization

Library   Math Functions / Matrices and Linear Algebra / Matrix Inverses

Description   The Cholesky Inverse block computes the inverse of the Hermitian positive definite input matrix S by performing Cholesky factorization.

```
Sym. Pos. Def.
    Inverse

   (Chol)
```
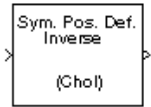
$$S^{-1} = (LL^*)^{-1}$$

L is a lower triangular square matrix with positive diagonal elements and L^* is the Hermitian (complex conjugate) transpose of *L*. Only the diagonal and upper triangle of the input matrix are used, and any imaginary component of the diagonal entries is disregarded. Cholesky factorization requires half the computation of Gaussian elimination (LU decomposition), and is always stable. The output is always sample based.

The algorithm requires that the input be Hermitian positive definite. When the input is not positive definite, the block reacts with the behavior specified by the **Non-positive definite input** parameter. The following options are available:

- Ignore — Proceed with the computation and *do not* issue an alert. The output is *not* a valid inverse.

- Warning — Display a warning message in the MATLAB Command Window, and continue the simulation. The output is *not* a valid inverse.

- Error — Display an error dialog box and terminate the simulation.

Note The **Non-positive definite input** parameter is a diagnostic parameter. Like all diagnostic parameters on the Configuration Parameters dialog box, it is set to Ignore in the Real-Time Workshop code generated for this block.

Dialog Box

Non-positive definite input
Response to nonpositive definite matrix inputs. Nontunable.

References Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

Supported Data Types
- Double-precision floating point
- Single-precision floating point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Cholesky Factorization	Signal Processing Blockset
Cholesky Solver	Signal Processing Blockset
LDL Inverse	Signal Processing Blockset
LU Inverse	Signal Processing Blockset
Pseudoinverse	Signal Processing Blockset
inv	MATLAB

See "Inverting Matrices" on page 6-10 for related information.

Cholesky Solver

Purpose   Solve *SX=B* for X when *S* is square Hermitian positive definite matrix

Library   Math Functions / Matrices and Linear Algebra / Linear System Solvers
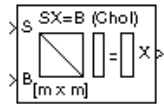
Description   The Cholesky Solver block solves the linear system *SX=B* by applying Cholesky factorization to input matrix at the S port, which must be square (*M*-by-*M*) and Hermitian positive definite. Only the diagonal and upper triangle of the matrix are used, and any imaginary component of the diagonal entries is disregarded. The input to the B port is the right side *M*-by-*N* matrix, *B*. The output is the unique solution of the equations, *M*-by-*N* matrix X, and is always sample based.

When the input is not positive definite, the block reacts with the behavior specified by the **Non-positive definite input** parameter. The following options are available:

- Ignore — Proceed with the computation and *do not* issue an alert. The output is *not* a valid solution.

- Warning — Proceed with the computation and display a warning message in the MATLAB Command Window. The output is *not* a valid solution.

- Error — Display an error dialog box and terminate the simulation.

Note The **Non-positive definite input** parameter is a diagnostic parameter. Like all diagnostic parameters on the Configuration Parameters dialog box, it is set to Ignore in the Real-Time Workshop code generated for this block.

A length-*M* vector input for right side *B* is treated as an *M*-by-1 matrix.

Algorithm   Cholesky factorization uniquely factors the Hermitian positive definite input matrix S as

$$S = LL^*$$

where *L* is a lower triangular square matrix with positive diagonal elements.

The equation *S*X=*B* then becomes

$$LL^*X = B$$

which is solved for X by making the substitution $Y = L * X$, and solving the following two triangular systems by forward and backward substitution, respectively.

$$LY = B$$

$$L^*X = Y$$

Dialog Box

Block Parameters: Cholesky Solver ✕

Cholesky Solver (mask)

Solve SX=B using Cholesky factorization. S must be a square, symmetric (Hermitian) positive definite matrix. B must have the same number of rows as S.

Parameters

Non-positive definite input: | Warning | ▼ |

| OK | Cancel | Help | Apply |
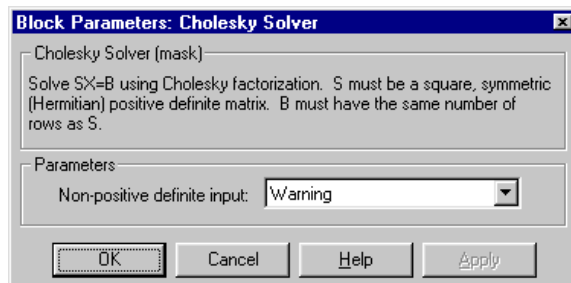
Non-positive definite input
　　Response to nonpositive definite matrix inputs. Nontunable.

Supported Data Types

- Double-precision floating point
- Single-precision floating point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

Cholesky Solver

See Also

Autocorrelation LPC	Signal Processing Blockset
Cholesky Factorization	Signal Processing Blockset
Cholesky Inverse	Signal Processing Blockset
LDL Solver	Signal Processing Blockset
LU Solver	Signal Processing Blockset
QR Solver	Signal Processing Blockset
chol	MATLAB

See "Solving Linear Systems" on page 6-7 for related information.

Purpose   Decimate signal using Cascaded Integrator-Comb filter

Library   Filtering / Multirate Filters

Description   The CIC Decimation block performs a sample rate decrease (decimation) on an input signal by an integer factor. Cascaded Integrator-Comb (CIC) filters are a class of linear phase FIR filters comprised of a comb section and an integrator section.
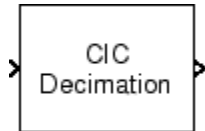
The transfer function of a CIC decimator filter is

$$H(z) = H_I^N(z)H_c^N(z) = \frac{\left(1 - z^{-RM}\right)^N}{\left(1 - z^{-1}\right)^N} = \left[\sum_{k=0}^{RM-1} z^{-k}\right]^N$$

where

- H_I is the transfer function of the integrator portion of the filter.

- H_C is the transfer function of the comb portion of the filter.

- N is the number of stages. The number of stages in a CIC filter is defined as the number of stages in either the comb section *or* the integrator section of the filter, not as the total number of stages throughout the entire filter.

- R is the decimation factor.

- M is the differential delay.

The CIC Decimation block supports real and complex fixed-point inputs. Each channel of a complex input is treated as two real input channels.

CIC Filter Structures

Some of the filter structures supported by the CIC Decimation and CIC Interpolation blocks exactly match those created by the mfilt CIC objects of the Filter Design Toolbox. If you have the Filter Design Toolbox installed, you can create an `mfilt` object in the MATLAB

CIC Decimation

workspace to specify in the **Multirate filter variable** parameter of this block. Otherwise, you can specify the CIC filter completely using only block dialog parameters. Using the block parameters also allows you to specify a resampler phase R for the filter, and to specify that the word length of the output should be the same as that of the final filter stage.

This block can be used to create either of the following CIC filter structures:

- "Decimator" on page 10-90
- "Zero-latency decimator" on page 10-90

Decimator

This decimator has a latency of N, where N is the number of stages in either the comb or the integrator section of the filter.
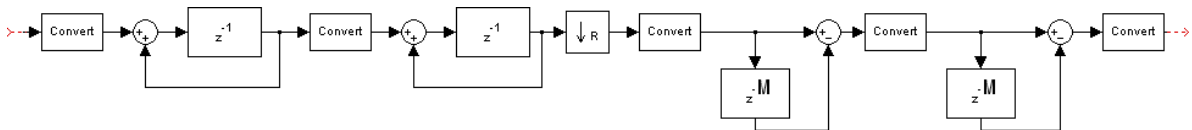
Zero-latency decimator

This filter is the classical Hogenauer CIC decimator, which has zero latency.
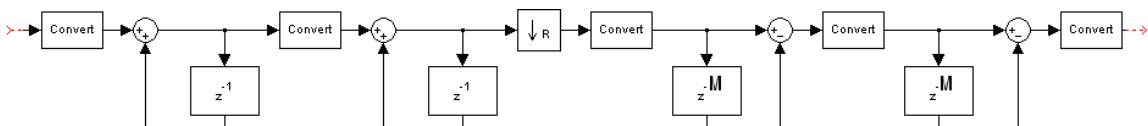
The word length and fraction length at each stage of the decimator are shown in the following table. You specify the word length for each filter stage in the **Word length for each of the (2*N) filter stages** parameter. The calculation for each fraction length is shown below:

Decimator Word Lengths and Fraction Lengths

	Word Length	Fraction Length
Filter Input	InputWL	InputFL
1st Stage Output	StageOneWL	InputFL
2nd Stage Output	StageTwoWL	InputFL + (StageTwoWL - StageOneWL)
3rd Stage Output	StageThreeWL	StageTwoFL + (StageThreeWL - StageTwoWL)
4th Stage Output	StageFourWL	StageThreeFL + (StageFourWL - StageThreeWL)
Nth Stage Output	Stage(N)WL	Stage(N-1)FL + (Stage(N)WL - Stage(N-1)WL)
Filter Output	OutputWL	FinalStageFL + (OutputWL - FinalStageWL)

Constraints and Conversions

The classic paper on CIC filters by Eugene B. Hogenauer, "An Economical Class of Digital Filters for Decimation and Interpolation" [3], describes the constraints on CIC decimator filters. These constraints are enforced by the CIC Decimation block.

CIC decimators have the following two constraints:

• The word lengths of the filter stages must be nonincreasing. That is, the word length of each filter stage must be the same size as, or smaller than, the word length of the previous filter stage.

• The number of bits of the first filter stage should be greater than or equal to the quantity B_{max}.

The formula for B_{max}, the most significant bit at the filter output, is given in the Hogenauer paper.

$$B_{max} = \text{ceil}(N\log_2 RM + B_{in} - 1)$$

CIC Decimation

where B_{in} is the number of bits of the input. You can choose whether the simulation will Warn, Error, or Ignore when the number of bits of the first filter stage is less than B_{max}.

The conversions denoted by the Convert blocks in the decimator diagrams in "CIC Filter Structures" on page 10-89 perform the changes between the word lengths of each stage. When you specify word lengths that do not follow the constraints described in this section, the block returns an error.
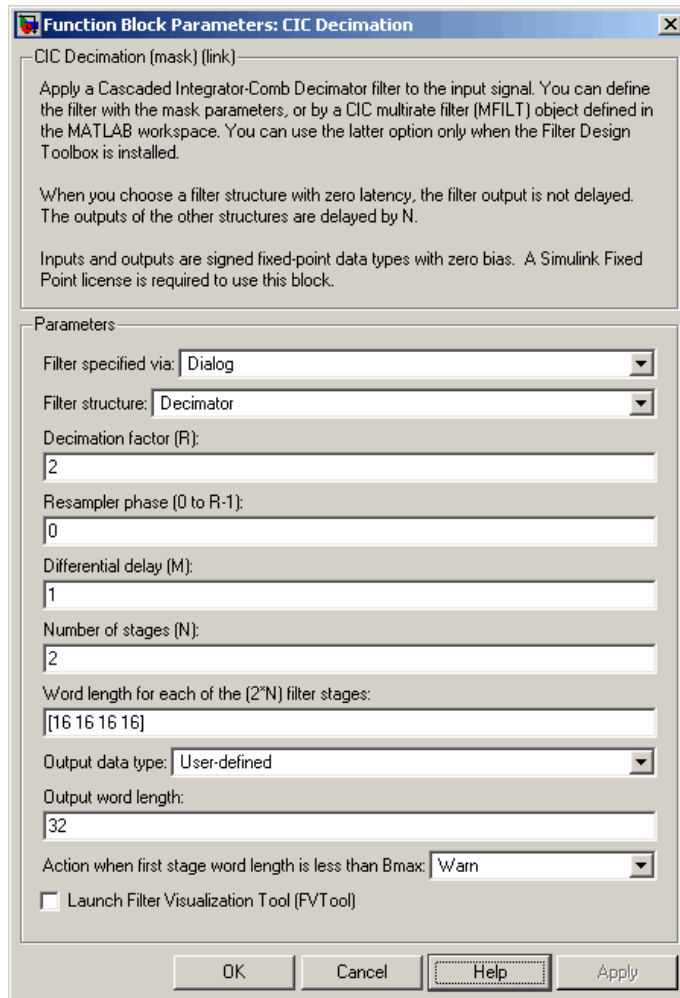
When you specify the word lengths correctly, the most significant bit B_{max} stays the same throughout the filter, while the word length of each stage either decreases or stays the same. This can cause the fraction length to change throughout the filter as least significant bits are truncated to decrease the word length, as shown in Decimator Word Lengths and Fraction Lengths on page 10-91.

When you select Same as final filter stage for the **Output data type** parameter, the final conversion shown in "CIC Filter Structures" on page 10-89 is not performed.

Sharing Models

You can share models that include CIC Decimation blocks among different people in your company, whether they have the Filter Design Toolbox installed or not. If you have the Filter Design Toolbox installed, you can configure a CIC Decimation block by selecting Multirate object in workspace for the **Filter specified via** parameter and entering a multirate filter object name in the **Multirate filter variable** parameter. As long as you run the model first, you can then share it with others who have the Signal Processing Blockset, even if they do not have the Filter Design Toolbox installed. When the Filter Design Toolbox is not installed, the block dialog parameters will automatically be populated with the values that correspond to the multirate filter object with which you ran the model previously.

Dialog Box

Function Block Parameters: CIC Decimation

CIC Decimation (mask) (link)

Apply a Cascaded Integrator-Comb Decimator filter to the input signal. You can define the filter with the mask parameters, or by a CIC multirate filter (MFILT) object defined in the MATLAB workspace. You can use the latter option only when the Filter Design Toolbox is installed.

When you choose a filter structure with zero latency, the filter output is not delayed. The outputs of the other structures are delayed by N.

Inputs and outputs are signed fixed-point data types with zero bias. A Simulink Fixed Point license is required to use this block.

Parameters

Filter specified via: Dialog

Filter structure: Decimator

Decimation factor (R):

2

Resampler phase (0 to R-1):

0

Differential delay (M):

1

Number of stages (N):

2

Word length for each of the (2*N) filter stages:

[16 16 16 16]

Output data type: User-defined

Output word length:

32

Action when first stage word length is less than Bmax: Warn

☐ Launch Filter Visualization Tool (FVTool)

OK   Cancel   Help   Apply

Filter specified via

Choose to design the CIC filter either through parameters in the dialog, or by defining a CIC multirate filter object (mfilt) in the

MATLAB workspace. This parameter is only visible when you have the Filter Design Toolbox installed.

Multirate filter variable

Specify the multirate filter variable that you have defined in the MATLAB workspace that defines the CIC filter you want to create. This parameter is only visible when you select `Multirate object in workspace` for the **Filter specified via** parameter.

The variable name you enter into this field must be the name of a filter object created with mfilt.cicdecim, a Filter Design Toolbox function.

Filter structure

Select one of the following CIC filter structures:

- Decimator — CIC decimator with latency *N*

- Zero-latency decimator — Classical Hogenauer CIC decimator with zero latency

Refer to "CIC Filter Structures" on page 10-89 for diagrams of these filter structures.

This parameter is only visible when you select `Dialog` for the **Filter specified via** parameter.

Decimation factor (R)

Specify the decimation factor of the filter.

This parameter is only visible when you select `Dialog` for the **Filter specified via** parameter.

Resampler phase (0 to R-1)

Specify the number of sample periods by which to delay the downsample operation internal to the CIC filter. The resampler phase must be an integer between 0 and 1 less than the decimation factor.

This parameter is only visible when you select `Dialog` for the **Filter specified via** parameter.

Differential delay (M)

Specify the differential delay of the comb portion of the filter, M, as shown in the diagrams in "CIC Filter Structures" on page 10-89.

This parameter is only visible when you select Dialog for the **Filter specified via** parameter.

Number of stages (N)

Specify the number of filter stages. This number is equal to the number of stages in either the comb portion of the filter or in the integrator portion of the filter. This value is not equal to the total number of stages in the comb and integrator sections combined.

This parameter is only visible when you select Dialog for the **Filter specified via** parameter.

Word length for each of the (2*N) filter stages

Enter the word length, in bits, of each of the stages of the filter. You need to enter a value for each stage in both the integrator and comb sections of the filter. Therefore, you need to enter a vector of length (2*N).

Alternately, you can enter a single integer when the word lengths of all the filter stages are the same.

The word lengths of the filter stages of a decimator must be nonincreasing. For more information about valid values for this parameter, refer to "Constraints and Conversions" on page 10-91.

Note The word length of the input signal to the block must be shorter than the word length that you designate for the first filter stage (in accordance with the equation for B_{max} in "Constraints and Conversions" on page 10-91). Otherwise, the block will return an error.

This parameter is only visible when you select `Dialog` for the **Filter specified via** parameter.

Output data type

Choose how you will specify the output word length and fraction length. Refer to the table Decimator Word Lengths and Fraction Lengths on page 10-91 for more information.

- When you select `Same as final filter stage`, the final conversion shown in the diagrams in "CIC Filter Structures" on page 10-89 is not performed. Therefore, the output word length is the same as the final filter stage word length, and the output fraction length is

  `Stage(N-1)FL + (Stage(N)WL - Stage(N-1)WL)`

  where `Stage(N)` is the final filter stage, as shown in the table Decimator Word Lengths and Fraction Lengths on page 10-91.

- When you select `Same as input`, these characteristics will match those of the block input.

- When you select `User-defined`, the **Output word length** parameter appears. You define the word length of the output, and the output fraction length is

  `FinalStageFL + (OutputWL - FinalStageWL)`

  as shown in Decimator Word Lengths and Fraction Lengths on page 10-91.

This parameter is only visible when you select `Dialog` for the **Filter specified via** parameter.

Output word length

Specify the word length of the output of the filter in bits. This parameter is only visible when you select `User-defined` for the **Output data type** parameter.

Action when first stage word length is less than Bmax

Choose whether the simulation will `Warn`, `Error`, or `Ignore` when the number of bits of the first filter stage is less than B_{max}.

Launch Filter Visualization Tool (FVTool)

Select to open the Filter Visualization Tool. Deselect to close FVTool. For more information on this GUI, refer to the `fvtool` reference page in the Signal Processing Toolbox User's Guide documentation.

This check box is only visible when you have the Filter Design Toolbox installed.

References

[1] Donadio, M., *Cascaded Integrator-Comb (CIC) Filter Introduction,* `http://www.dspguru.com/info/tutor/cic.htm`

[2] Frerking, Marvin E., *Digital Signal Processing in Communications Systems*, Kluwer Academic Publishers, 1994.

[3] Hogenauer, E.B., "An Economical Class of Digital Filters for Decimation and Interpolation," *IEEE Transactions on Acoustics, Speech and Signal Processing*, ASSP-29(2): pp. 155-162, 1981.

[4] *LogicCore Cascaded Integrator-Comb (CIC) Filter V3.0* (product specification) `http://www.xilinx.com/ipcenter/catalog/logicore/docs/cic.pdf`

[5] Meyer-Baese, U., *Digital Signal Processing with Field Programmable Gate Arrays,* Springer Verlag, 2001.

Supported Data Types

- Fixed point (signed only)

- 8-, 16-, and 32-bit signed integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

CIC Interpolation	Signal Processing Blockset
FIR Decimation	Signal Processing Blockset

CIC Decimation

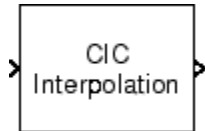|	
FIR Interpolation	Signal Processing Blockset
`filter`	Filter Design Toolbox
`mfilt.cicdecim`	Filter Design Toolbox
`mfilt.cicinterp`	Filter Design Toolbox

Purpose   Interpolate signal using Cascaded Integrator-Comb filter

Library   Filtering / Multirate Filters

Description The CIC Interpolation block performs a sample rate increase
(interpolation) on an input signal by an integer factor. Cascaded
Integrator-Comb (CIC) filters are a class of linear phase FIR filters
comprised of a comb section and an integrator section.

The transfer function of a CIC interpolator filter is

$$H(z) = H_I{}^N(z)\, H_C{}^N(z) = \frac{(1 - z^{-RM})^N}{(1 - z^{-1})^N} = \left[\sum_{k=0}^{RM-1} z^{-k} \right]^N$$

where

- H_I is the transfer function of the integrator portion of the filter.

- H_C is the transfer function of the comb portion of the filter.

- N is the number of stages. The number of stages in a CIC filter is
  defined as the number of stages in either the comb section *or* the
  integrator section of the filter, not as the total number of stages
  throughout the entire filter.

- R is the interpolation factor.

- M is the differential delay.

The CIC Interpolation block supports real and complex fixed-point
inputs. Each channel of a complex input is treated as two real input
channels.

CIC Filter Structures

Some of the filter structures supported by the CIC Interpolation and
CIC Decimation blocks exactly match those created by the mfilt CIC
objects of the Filter Design Toolbox. If you have the Filter Design
Toolbox installed, you can create an `mfilt` object in the MATLAB
workspace to specify in the **Multirate filter variable** parameter of this
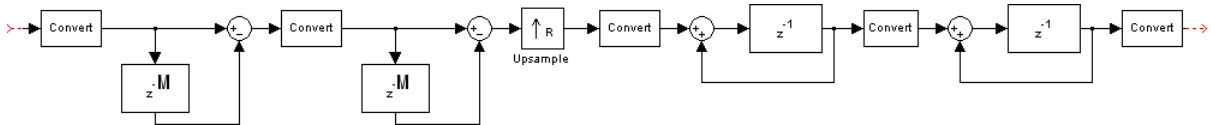
CIC Interpolation

block. Otherwise, you can specify the CIC filter completely using only block dialog parameters. Using the block parameters also allows you to specify a resampler phase R for the filter, and to specify that the word length of the output should be the same as that of the final filter stage.

This block can be used to create either of the following CIC filter structures:

- "Interpolator" on page 10-100
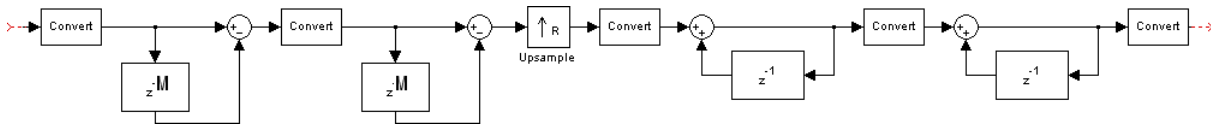- "Zero-latency interpolator" on page 10-100

Interpolator

This interpolator has a latency of N, where N is the number of stages in either the comb or the integrator section of the filter.

Zero-latency interpolator

This filter is the classical Hogenauer CIC interpolator, which has zero latency.

The word length and fraction length at each stage of the interpolator are shown in the following table. When you select User-defined for the **Filter stages word length mode parameter**, you specify the word length for each filter stage in the **Word length for each of the (2*N) filter stages** parameter. When you select Use minimum number of bits per stage, the filter stage word lengths are automatically set to the minimum number of bits possible in a valid CIC interpolator, in

accordance with the formula for W_j in "Constraints and Conversions" on page 10-101. The calculation for each fraction length is shown below:

Interpolator Word Lengths and Fraction Lengths

	Word Length	Fraction Length
Filter Input	InputWL	InputFL
1st Stage Output	StageOneWL	InputFL
Nth Stage Output	Stage(N)WL	InputFL
Filter Output	OutputWL	InputFL + (OutputWL - FinalStageWL)

Constraints and Conversions

The classic paper on CIC filters by Eugene B. Hogenauer, "An Economical Class of Digital Filters for Decimation and Interpolation" [3], describes the constraints on CIC interpolator filters. These constraints are enforced by the CIC Interpolation block.

CIC interpolators have the following two constraints:

• The word lengths of the filter stages must be nondecreasing. That is, the word length of each filter stage must be the same size as, or greater than, the word length of the previous filter stage.

• The word length of each filter stage must be greater than or equal to the quantity W_j.

The formula for W_j, the minimum register width, is derived in the Hogenauer paper. The formula for W_j is given by

$$W_j = \text{ceil}(B_{in} + \log_2 G_j)$$

where G_j, the maximum register growth up to the jth stage, is given by

CIC Interpolation

$$G_j = \begin{cases} 2^j, & j = 1, 2, ..., N \\ \dfrac{2^{2N-j}(RM)^{j-N}}{R}, & j = N+1, ..., 2N \end{cases}$$

When the differential delay, M, is 1, there is also a special condition for the register width of the last comb, W_N, that is given by

$$W_N = B_{in} + N - 1 \qquad \text{if } M = 1$$

The conversions denoted by the Convert blocks in the integrator diagrams in "CIC Filter Structures" on page 10-89 perform the changes between the word lengths of each stage. When you specify word lengths that do not follow the constraints described in this section, the block returns an error.
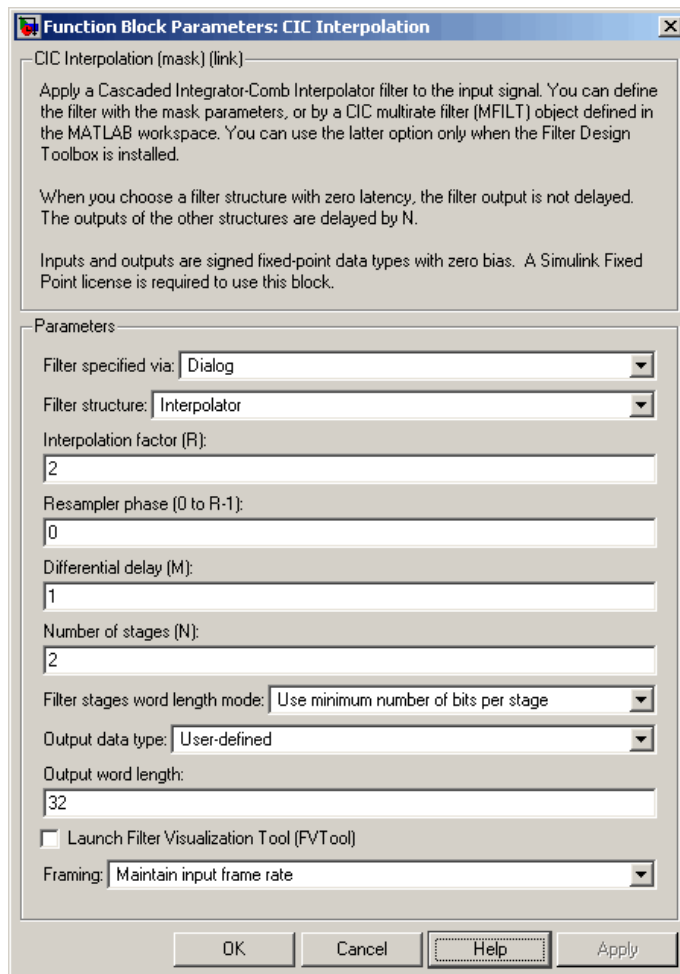
The fraction lengths and scalings of the filter stages will not change, because at each stage the word length is either staying the same or increasing. The scaling of the signal can change at the output after the final filter stage if you choose the output word length to be less than the word length of the final filter stage.

When you select `Same as final filter stage` for the **Output data type** parameter, the final conversion shown in "CIC Filter Structures" on page 10-99 is not performed.

Sharing Models

You can share models that include CIC Interpolation blocks among different people in your company, whether they have the Filter Design Toolbox installed or not. If you have the Filter Design Toolbox installed, you can configure a CIC Interpolation block by selecting `Multirate object in workspace` for the **Filter specified via** parameter and entering a multirate filter object name in the **Multirate filter variable** parameter. As long as you run the model first, you can then share it with others who have the Signal Processing Blockset, even if they do not have the Filter Design Toolbox installed. When the Filter Design Toolbox is not installed, the block dialog parameters will automatically be populated with the values that correspond to the multirate filter object with which you ran the model previously.

**Dialog
Box**

Function Block Parameters: CIC Interpolation

CIC Interpolation (mask) (link)

Apply a Cascaded Integrator-Comb Interpolator filter to the input signal. You can define the filter with the mask parameters, or by a CIC multirate filter (MFILT) object defined in the MATLAB workspace. You can use the latter option only when the Filter Design Toolbox is installed.

When you choose a filter structure with zero latency, the filter output is not delayed. The outputs of the other structures are delayed by N.

Inputs and outputs are signed fixed-point data types with zero bias. A Simulink Fixed Point license is required to use this block.

Parameters

Filter specified via: Dialog

Filter structure: Interpolator

Interpolation factor (R):
2

Resampler phase (0 to R-1):
0

Differential delay (M):
1

Number of stages (N):
2

Filter stages word length mode: Use minimum number of bits per stage

Output data type: User-defined

Output word length:
32

☐ Launch Filter Visualization Tool (FVTool)

Framing: Maintain input frame rate

[OK] [Cancel] [Help] [Apply]

Filter specified via

Choose to design the CIC filter either through parameters in the dialog, or by defining a CIC multirate filter object (mfilt) in the MATLAB workspace. This parameter is only visible when you have the Filter Design Toolbox installed.

CIC Interpolation

Multirate filter variable

Specify the multirate filter variable that you have defined in the MATLAB workspace that defines the CIC filter you want to create. This parameter is only visible when you select `Multirate object in workspace` for the **Filter specified via** parameter.

The variable name you enter into this field must be the name of a filter object created with mfilt.cicinterp, a Filter Design Toolbox function.

Filter structure

Select one of the following CIC filter structures:

- Interpolator — CIC interpolator with latency *N*

- Zero-latency interpolator — Classical Hogenauer CIC interpolator with zero latency

Refer to "CIC Filter Structures" on page 10-99 for diagrams of these filter structures.

This parameter is only visible when you select `Dialog` for the **Filter specified via** parameter.

Interpolation factor (R)

Specify the interpolation factor of the filter.

This parameter is only visible when you select `Dialog` for the **Filter specified via** parameter.

Resampler phase (0 to R-1)

Specify the number of sample periods by which to delay the upsample operation internal to the CIC filter. The resampler phase must be an integer between 0 and 1 less than the interpolation factor.

This parameter is only visible when you select `Dialog` for the **Filter specified via** parameter.

Differential delay (M)

Specify the differential delay of the comb portion of the filter, M, as shown in the diagrams in "CIC Filter Structures" on page 10-89.

This parameter is only visible when you select Dialog for the **Filter specified via** parameter.

Number of stages (N)

Specify the number of filter stages. This number is equal to the number of stages in either the comb portion of the filter or in the integrator portion of the filter. This value is not equal to the total number of stages in the comb and integrator sections combined.

This parameter is only visible when you select Dialog for the **Filter specified via** parameter.

Filter stages word length mode

Specify the method by which the word lengths of the filter stages are determined.

- When you select Use minimum number of bits per stage, the filter stage word lengths are automatically set to the minimum number of bits possible in a valid CIC interpolator, in accordance with the formula for W_j in "Constraints and Conversions" on page 10-101.

- When you select User-defined, the **Word length for each of the (2*N) filter stages** parameter appears.

This parameter is only visible when you select Dialog for the **Filter specified via** parameter.

Word length for each of the (2*N) filter stages

Enter the word length, in bits, of each of the stages of the filter. You need to enter a value for each stage in both the integrator and comb sections of the filter. Therefore, you need to enter a vector of length (2*N).

Alternately, you can enter a single integer if the word lengths of all the filter stages are the same.

The word lengths of the filter stages of an interpolator must be nondecreasing. For more information about valid values for this parameter, refer to "Constraints and Conversions" on page 10-91.

Note The word length of the input signal to the block must be shorter than the word length that you designate for the first filter stage (in accordance with the formula for W_j in "Constraints and Conversions" on page 10-101). Otherwise, the block will return an error.

This parameter is only visible when you select Dialog for the **Filter specified via** parameter, and when you select User-defined for the **Filter stages word length mode** parameter.

Output data type

Choose how you will specify the output word length and fraction length. Refer to the table Interpolator Word Lengths and Fraction Lengths on page 10-101 for more information.

- When you select Same as final filter stage, the final conversion shown in the diagrams in "CIC Filter Structures" on page 10-99 is not performed. Therefore, the output word length is the same as the final filter stage word length, and the output fraction length is the input fraction length, as shown in the table Interpolator Word Lengths and Fraction Lengths on page 10-101.

- When you select Same as input, these characteristics will match those of the block input.

- When you select User-defined, the **Output word length** parameter appears. You define the word length of the output, and the output fraction length is

```
InputFL + (OutputWL - FinalStageWL)
```

as shown in Interpolator Word Lengths and Fraction Lengths on page 10-101.

This parameter is only visible when you select `Dialog` for the **Filter specified via** parameter.

Output word length

Specify the word length of the output of the filter in bits. This parameter is only visible when you select `User-defined` for the **Output data type** parameter.

Launch Filter Visualization Tool (FVTool)

Select to open the Filter Visualization Tool. Deselect to close FVTool. For more information on this GUI, refer to the `fvtool` reference page in the Signal Processing Toolbox User's Guide documentation.

This check box is only visible when you have the Filter Design Toolbox installed.

Framing

Specify the method by which to implement the interpolation for frame-based inputs, `Maintain input frame size` or `Maintain input frame rate`.

References

[1] Donadio, M., *Cascaded Integrator-Comb (CIC) Filter Introduction,* http://www.dspguru.com/info/tutor/cic.htm

[2] Frerking, Marvin E., *Digital Signal Processing in Communications Systems*, Kluwer Academic Publishers, 1994.

[3] Hogenauer, E.B., "An Economical Class of Digital Filters for Decimation and Interpolation," *IEEE Transactions on Acoustics, Speech and Signal Processing*, ASSP-29(2): pp. 155-162, 1981.

[4] *LogicCore Cascaded Integrator-Comb (CIC) Filter V3.0*(product specification) http://www.xilinx.com/ipcenter/catalog/logicore/docs/cic.pdf

CIC Interpolation

[5] Meyer-Baese, U., *Digital Signal Processing with Field Programmable Gate Arrays*, Springer Verlag, 2001.

Supported Data Types

- Fixed point (signed only)
- 8-, 16-, and 32-bit signed integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

CIC Decimation	Signal Processing Blockset
FIR Decimation	Signal Processing Blockset
FIR Interpolation	Signal Processing Blockset
filter	Filter Design Toolbox
mfilt.cicdecim	Filter Design Toolbox
mfilt.cicinterp	Filter Design Toolbox

Purpose   Compute complex cepstrum of input

Library   Transforms
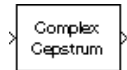
Description   The Complex Cepstrum block computes the complex cepstrum of each channel in the real-valued *M*-by-*N* input matrix, u. For both sample-based and frame-based inputs, the block assumes that each input column is a frame containing *M* consecutive samples from an independent channel. The block does not accept complex-valued inputs.

The input is altered by the application of a linear phase term so that there is no phase discontinuity at $\pm\pi$ radians. That is, each input channel is independently zero padded and circularly shifted to have zero phase at π radians.

The output is a real M_0-by-*N* matrix, where M_0 is specified by the **FFT length** parameter. Each output column contains the length-M_0 complex cepstrum of the corresponding input column.

```
y = cceps(u,Mo)     % Equivalent MATLAB code
```

When you select the **Inherit FFT length from input port dimensions** check box, the output frame size matches the input frame size (M_0=*M*). In this case, a *sample-based* length-*M* row vector input is processed as a single channel (that is, as an *M*-by-1 column vector), and the output is a length-*M* row vector. A 1-D vector input is *always* processed as a single channel, and the output is a 1-D vector.

The output is always sample based, and the output port rate is the same as the input port rate.

Complex Cepstrum

Dialog Box

Inherit FFT length from input port dimensions
When selected, matches the output frame size to the input frame size.

FFT length
The number of frequency points at which to compute the FFT, which is also the output frame size, M_o. This parameter is available when you do not select **Inherit FFT length from input port dimensions**.

Supported Data Types

- Double-precision floating point
- Single-precision floating point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

DCT	Signal Processing Blockset
FFT	Signal Processing Blockset

| Real Cepstrum | Signal Processing Blockset |
| cceps | Signal Processing Toolbox |

Complex Exponential

Purpose   Compute complex exponential function

Library   Math Functions / Math Operations

Description   The Complex Exponential block computes the complex exponential function for each element of the real input, *u*.
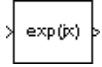
$$y = e^{ju} = \cos u + j\sin u$$

where $j = \sqrt{-1}$. The output is complex, with the same size and frame status as the input.
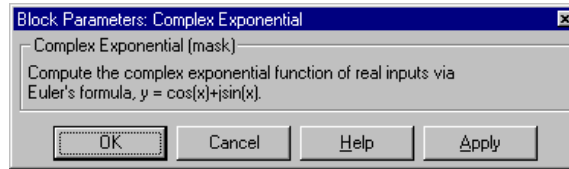
**Dialog
Box**

Block Parameters: Complex Exponential

Complex Exponential (mask)

Compute the complex exponential function of real inputs via
Euler's formula, y = cos(x)+jsin(x).

OK   Cancel   Help   Apply

**Supported
Data
Types**

- Double-precision floating point
- Single-precision floating point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Math Function	Simulink
Sine Wave	Signal Processing Blockset
exp	MATLAB

Purpose   Generate square, diagonal matrix

Library   • Signal Processing Sources

                   • Math Functions / Matrices and Linear Algebra / Matrix Operations

Description   The Constant Diagonal Matrix block outputs a square diagonal matrix
                   constant. The **Constant along diagonal** parameter determines the
                   values along the matrix diagonal. This parameter can be a scalar to be
                   repeated for all elements along the diagonal, or a vector containing the
                   values of the diagonal elements. To generate the identity matrix, set
                   the **Constant along diagonal** to 1, or use the Identity Matrix block.

                   The output is frame based when you select the **Frame-based output**
                   check box; otherwise, the output is sample based.

Dialog   The **Main** pane of the Constant Diagonal Matrix block dialog appears
Box   as follows:
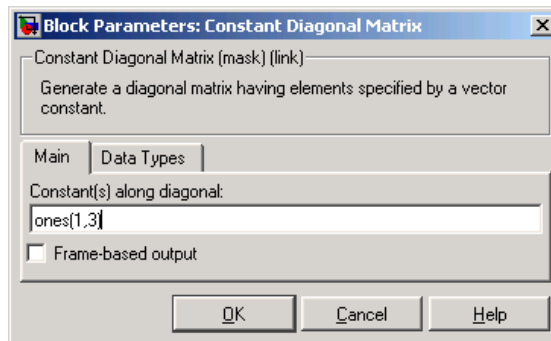
                   Opening this dialog box causes a running simulation to pause.
                   See "Changing Source Block Parameters" in the online Simulink
                   documentation for details.

                   Constant(s) along diagonal
                        Specify the values of the elements along the diagonal. You can
                        input a scalar or a vector. Tunable.

Constant Diagonal Matrix

When you specify any data type information in this field, it is overridden by the value of the **Output data type** parameter on the **Data Types** pane, unless you select Inherit from 'Constant(s) along diagonal'.

Frame-based output
Select to cause the output of the block to be frame based. Otherwise, the output is sample based.

The **Data types** pane of the Constant Diagonal Matrix block dialog appears as follows:
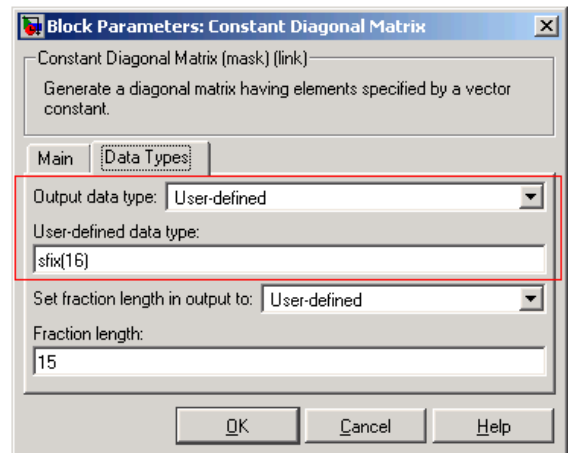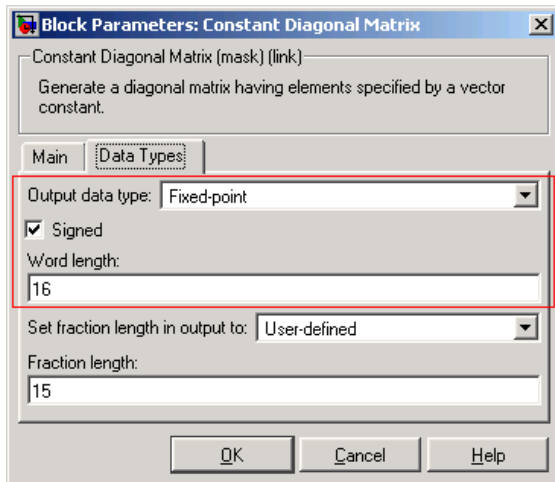
Output data type
Specify the output data type in one of the following ways:

- Choose one of the built-in data types from the list.

- Choose Fixed-point to specify the output data type and scaling in the **Signed**, **Word length**, **Set fraction length in output to,** and **Fraction length** parameters.

- Choose `User-defined` to specify the output data type and scaling in the **User-defined data type**, **Set fraction length in output to,** and **Fraction length** parameters.

- Choose `Inherit from 'Constant(s) along diagonal'` to set the output data type and scaling to match the values of the **Constant(s) along diagonal** parameter on the **Main** pane.

- Choose `Inherit via back propagation` to set the output data type and scaling to match the next block downstream.

The value of this parameter overrides any data type information specified in the **Constant(s) along diagonal** parameter on the **Main** pane, except when you select `Inherit from 'Constant(s) along diagonal'`.

Signed

Select to output a signed fixed-point signal. Otherwise, the signal will be unsigned. This parameter is only visible when you select `Fixed-point` for the **Output data type** parameter.

Word length

Specify the word length, in bits, of the fixed-point output data type. This parameter is only visible when you select `Fixed-point` for the **Output data type** parameter.

User-defined data type

Specify any built-in or fixed-point data type. You can specify fixed-point data types using the `sfix`, `ufix`, `sint`, `uint`, `sfrac`, and `ufrac` functions from Simulink Fixed Point. This parameter is only visible when you select `User-defined` for the **Output data type** parameter.

Set fraction length in output to

Specify the scaling of the fixed-point output by either of the following two methods:

- Choose `Best precision` to have the output scaling automatically set such that the output signal has the best possible precision.

- Choose `User-defined` to specify the output scaling in the **Fraction length** parameter.

  This parameter is only visible when you select `Fixed-point` for the **Output data type** parameter, or when you select `User-defined` and the specified output data type is a fixed-point data type.

Fraction length

For fixed-point output data types, specify the number of fractional bits, or bits to the right of the binary point. This parameter is only visible when you select `Fixed-point` or `User-defined` for the **Output data type** parameter and `User-defined` for the **Set fraction length in output to** parameter.

Supported Data Types

- Double-precision floating point

- Single-precision floating point

- Fixed point

- 8-, 16-, and 32-bit signed integers

- 8-, 16-, and 32-bit unsigned integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Create Diagonal Matrix	Signal Processing Blockset
DSP Constant	Signal Processing Blockset
Identity Matrix	Signal Processing Blockset
diag	MATLAB

Purpose   Generate ramp signal with length based on input dimensions

Library   Signal Operations

Description   The Constant Ramp block generates the constant ramp signal

```
y = (0:L-1)*m + b
```

where m is the slope specified by the scalar **Slope** parameter, and b is the *y*-intercept specified by the scalar **Offset** parameter.

For a matrix input, the length L of the output ramp is equal to either the number of rows or the number of columns in the input, as determined by the **Ramp length equals number of** parameter. For a 1-D vector input, L is equal to the length of the input vector. The output, y, is always a 1-D vector.

Dialog Box   The **Main** pane of the Constant Ramp block dialog appears as follows:
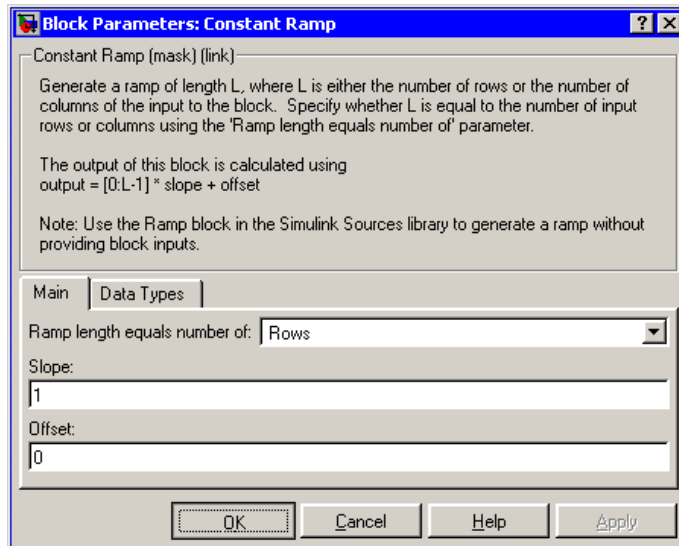
Ramp length equals number of

Specify the dimension of the input matrix that determines the length of the output ramp, Rows or Columns.

Slope

Specify the scalar slope of the ramp.

Offset

Specify the scalar *y*-intercept of the ramp.

The **Data types** pane of the Constant Ramp block dialog appears as follows:
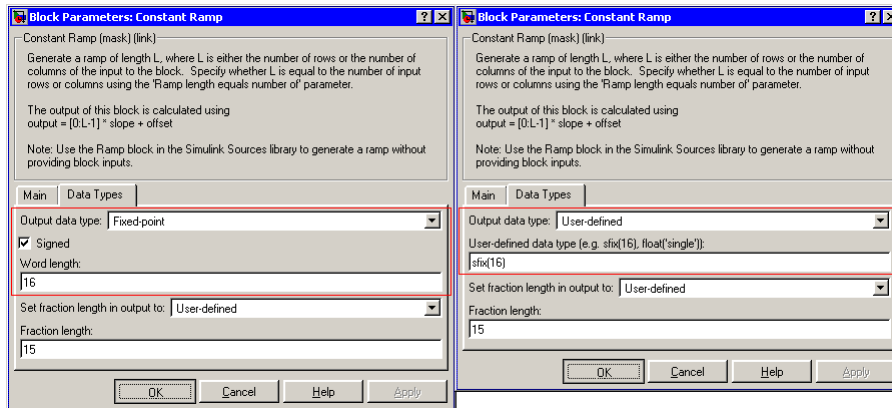
Output data type

Specify the output data type in one of the following ways:

- Choose Same as input to force the data type of the output to be the same as the data type of the input to the block.

- Choose one of the built-in data types from the list.

- Choose Fixed-point to specify the output data type and scaling in the **Signed**, **Word length**, **Set fraction length in output to,** and **Fraction length** parameters.

- Choose `User-defined` to specify the output data type and scaling in the **User-defined data type**, **Set fraction length in output to,** and **Fraction length** parameters.

- Choose `Inherit via back propagation` to set the output data type and scaling to match the next block downstream.

This block differs from other Signal Processing Blockset blocks in that unless you choose `Same as input` for this parameter, the data types of the input and the output do not need to be the same.

Signed

Select to output a signed fixed-point signal. Otherwise, the signal will be unsigned. This parameter is only visible when you select `Fixed-point` for the **Output data type** parameter.

Word length

Specify the word length, in bits, of the fixed-point output data type. This parameter is only visible when you select `Fixed-point` for the **Output data type** parameter.

User-defined data type

Specify any built-in or fixed-point data type. You can specify fixed-point data types using the `sfix`, `ufix`, `sint`, `uint`, `sfrac`, and `ufrac` functions from Simulink Fixed Point. This parameter is only visible when you select `User-defined` for the **Output data type** parameter.

Set fraction length in output to

Specify the scaling of the fixed-point output by either of the following two methods:

- Choose `Best precision` to have the output scaling automatically set such that the output signal has the best possible precision.

- Choose `User-defined` to specify the output scaling in the **Fraction length** parameter.

This parameter is only visible when you select `Fixed-point` for the **Output data type** parameter, or when you select `User-defined` and the specified output data type is a fixed-point data type.

Fraction length

For fixed-point output data types, specify the number of fractional bits, or bits to the right of the binary point. This parameter is only visible when you select `Fixed-point` or `User-defined` for the **Output data type** parameter and `User-defined` for the **Set fraction length in output to** parameter.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

This block differs from other Signal Processing Blockset blocks in that unless you choose `Same as input` for the **Output data type** parameter, the data types of the input and the output do not need to be the same.

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Create Diagonal Matrix	Signal Processing Blockset
DSP Constant	Signal Processing Blockset
Identity Matrix	Signal Processing Blockset

Purpose   Create a discontiguous input in a contiguous block of memory for
Real-Time Workshop code generated from blocks linked to versions
of the DSP Blockset prior to 4.0

Library   dspobslib

Description
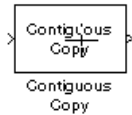
> **Note** The Contiguous Copy block is still supported but is likely to be
> obsoleted in a future release.

The Contiguous Copy block copies the input to a contiguous block of
memory, and passes this new copy to the output. The output is identical
to the input, but is guaranteed to reside in a contiguous section of
memory.

Because Simulink employs an efficient copy-by-reference method for
propagating data in a model, some operations produce outputs with
discontiguous memory locations.

Although this does not present a problem during simulation, blocks
linked to versions of the DSP Blockset prior to 4.0 may require
contiguous inputs for code generation with Real-Time Workshop. When
such blocks are used in a model intended for code generation, they
should be preceded by the Contiguous Copy block to ensure that their
inputs are contiguous.

**Dialog
Box**
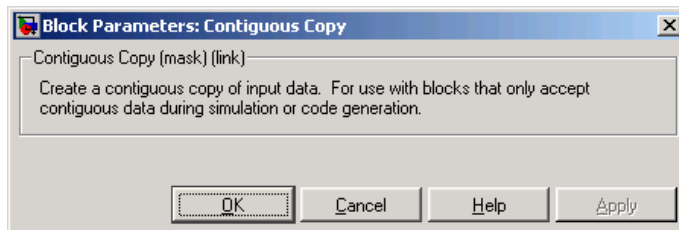
Contiguous Copy

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point
- Custom data types
- Boolean
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

Purpose   Reshape 1-D or 2-D input to a 2-D matrix with specified dimensions

Library   Signal Management / Signal Attributes
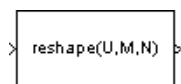
Description   The Convert 1-D to 2-D block reshapes a length-M_i 1-D vector or
an M_i-by-N_i matrix to an M_o-by-N_o matrix, where M_o is specified by
the **Number of output rows** parameter, and N_o is specified by the
Number of output columns parameter.

```
y = reshape(u,Mo,No)        % Equivalent MATLAB code
```

The input is reshaped *columnwise*, as shown in the two cases below.
The length-6 vector and the 2-by-3 matrix are both reshaped to the
same 3-by-2 output matrix.
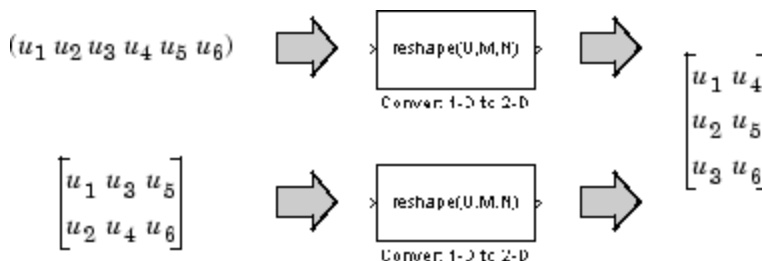
An error is generated when $(M_o*N_o) \neq (M_i*N_i)$. That is, the total number
of input elements must be conserved in the output.

The output is frame based when you select the **Frame-based output**
check box; otherwise, the output is sample based.

Convert 1-D to 2-D

Dialog Box
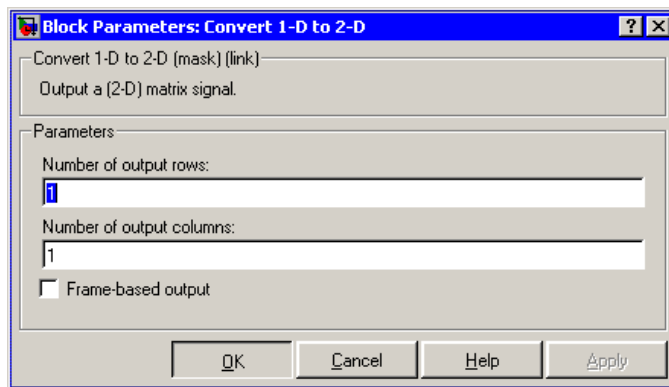
Number of output rows
>   The number of rows, M_o, in the output matrix.

Number of output columns
>   The number of rows, N_o, in the output matrix.

Frame-based output
>   Creates a frame-based output when selected.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Custom data types • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Custom data types • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Buffer	Signal Processing Blockset
Convert 2-D to 1-D	Signal Processing Blockset
Frame Status Conversion	Signal Processing Blockset
Reshape	Simulink
Submatrix	Signal Processing Blockset

Convert 2-D to 1-D

Purpose Convert 2-D matrix input to 1-D vector

Library Signal Management / Signal Attributes

Description The Convert 2-D to 1-D block reshapes an *M*-by-*N* matrix input to a 1-D vector with length *M*N*.

```
y = u(:)     % Equivalent MATLAB code
```

The input is reshaped *columnwise*, as shown below for a 3-by-2 matrix.

$$\begin{bmatrix} u_1 & u_4 \\ u_2 & u_5 \\ u_3 & u_6 \end{bmatrix} \Rightarrow \boxed{\text{U(:)}} \Rightarrow (u_1 \ u_2 \ u_3 \ u_4 \ u_5 \ u_6)$$

Convert 2-D to 1-D

The output is always sample-based.

Dialog Box

Block Parameters: Convert 2-D to 1-D

Convert 2-D to 1-D (mask) (link)

Output a (1-D) vector signal.

| OK | Cancel | Help | Apply |

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Custom data types • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Custom data types • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Buffer	Signal Processing Blockset
Convert 1-D to 2-D	Signal Processing Blockset
Frame Status Conversion	Signal Processing Blockset

Convert 2-D to 1-D

| Reshape | Simulink |
| Submatrix | Signal Processing Blockset |

Purpose   Compute convolution of two inputs

Library   Signal Operations

Description   The Convolution block mathematically convolves analogous columns of
an M_u-by-N input matrix u and an M_v-by-N input matrix v. This block
can also independently convolve a single-channel column vector with
each channel of a multiple-channel matrix.

The Convolution block does not accept sample-based full-dimension
matrix inputs, or mixed sample-based row vector and column vector
inputs. All outputs are sample based.

The Convolution block accepts both real and complex floating-point
and fixed-point inputs. Fixed-point signals are not supported for the
frequency domain.

Convolving Frame-Based Inputs

Matrix inputs to the Convolution block must be frame based. The
output, y, is a frame-based (M_u+M_v-1)-by-N matrix whose jth column
has elements

$$y_{i(\ ),j} = \sum_{k=1}^{max(M_u, M_v)} u_{k,j} v_{(i-k+1),j} \qquad 1 \le i \le (M_u + M_v - 1)$$

Inputs u and v are zero when indexed outside of their valid ranges.
When both inputs are real, the output is real; when one or both inputs
are complex, the output is complex.

When one input is a column vector (single channel) and the other is a
matrix (multiple channels), the single-channel input is independently
convolved with each channel of the multichannel input. For example,
when u is a M_u-by-1 column vector and v is an M_v-by-N matrix, the
output is an (M_u+M_v-1)-by-N matrix whose jth column has elements

$$y_{i,j} = \sum_{k=1}^{max(M_u, M_v)} u_k v_{(i-k+1),j} \qquad 1 \le i \le (M_u + M_v - 1)$$

Convolution

Convolving Sample-Based Inputs

When u and v are sample-based vectors with lengths M_u and M_v, the Convolution block performs the vector convolution

$$y_i = \sum_{k=1}^{max(M_u, M_v)} u_k v_{(i-k+1)} \qquad 1 \leq i \leq (M_u + M_v - 1)$$

The dimensions of the sample-based output vector are determined by the dimensions of the input vectors:

- When both inputs are row vectors, or when one input is a row vector and the other is a 1-D vector, the output is a 1-by-(M_u+M_v-1) row vector.

- When both inputs are column vectors, or when one input is a column vector and the other is a 1-D vector, the output is a (M_u+M_v-1)-by-1 column vector.

- When both inputs are 1-D vectors, the output is a 1-D vector of length M_u+M_v-1.

Fixed-Point Data Types

The following diagram shows the data types used within the Convolution block for fixed-point signals (time domain only).
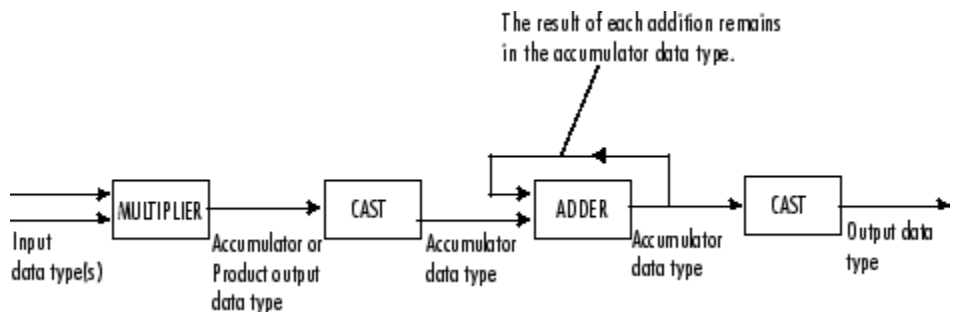
You can set the product output, accumulator, and output data types in the block dialog as discussed below.

The output of the multiplier is in the product output data type when the input is real. When the input is complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed, refer to "Multiplication Data Types" on page 8-16.

Dialog Box

The **Main** pane of the Convolution block dialog appears as follows:
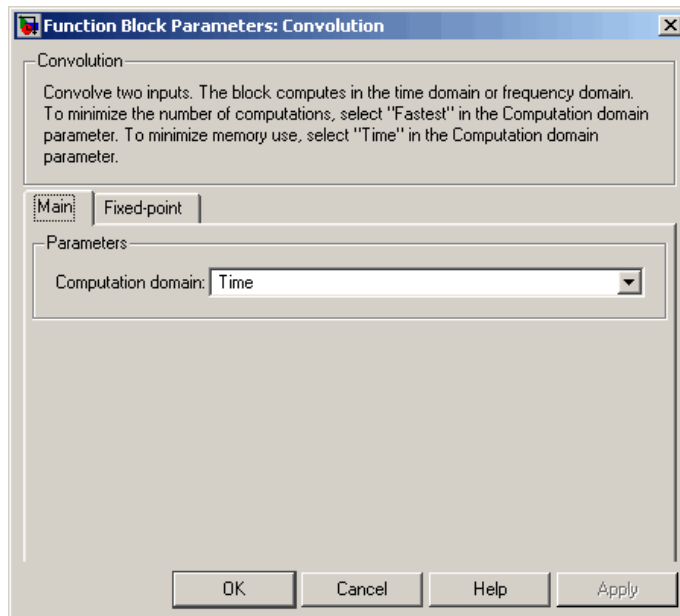
Computation domain

Set the domain in which the block computes convolutions:

- Time — The block computes in the time domain, which minimizes memory use.

- `Frequency` — The block computes in the frequency domain, which might require fewer computations than computing in the time domain, depending on the input length.

- `Fastest` — The block computes in the domain, which minimizes the number of computations.

The **Fixed-point** pane of the Convolution block dialog appears as follows:

Note Fixed-point signals are only supported for the time domain. To use the parameters on this pane, make sure `Time` is selected for the **Computation domain** parameter on the **Main** pane.

Rounding mode

Select the rounding mode for fixed-point operations.

Overflow mode

Select the overflow mode for fixed-point operations.

Product output

Use this parameter to specify how you would like to designate the product output word and fraction lengths. Refer to "Fixed-Point Data Types" on page 10-8 and "Multiplication Data Types" on page 8-16 for illustrations depicting the use of the product output data type in this block:

- When you select Inherit via internal rule, the product output word length and fraction length are automatically set according to the following equations:

$$ideal\ product\ output\ word\ length\ =\ word\ length\ of\ first\ input\ +\ word\ length\ of\ second\ input$$

$$ideal\ product\ output\ fraction\ length\ =\ fraction\ length\ of\ first\ input\ +\ fraction\ length\ of\ second\ input$$

Note The actual product output word length may be equal to or greater than the calculated ideal product output word length, depending on the settings on the **Hardware Implementation** pane of the Configuration Parameters dialog box.

- When you select Same as first input, these characteristics will match those of the first input to the block.

- When you select Binary point scaling, you are able to enter the word length and the fraction length of the product output, in bits.

- When you select Slope and bias scaling, you are able to enter the word length, in bits, and the slope of the product

output. This block requires power-of-two slope and a bias of
zero.

Accumulator

The result of each addition remains
in the accumulator data type.

As depicted above, inputs to the accumulator are cast to the
accumulator data type. The output of the adder remains in the
accumulator data type as each element of the input is added to
it. Use this parameter to specify how you would like to designate
this accumulator word and fraction lengths.

You also use this parameter to specify the accumulator word and
fraction lengths resulting from a complex-complex multiplication
in the block. Refer to "Multiplication Data Types" on page 8-16 for
more information.

• When you select Inherit via internal rule, the
  accumulator word length and fraction length are automatically
  set according to the following equations:

  If at least one of the inputs is real:

  $$ideal\ accumulator\ word\ length = ideal\ product\ output\ word\ length + \text{floor}(\log 2(\text{MIN}$$
  $$((number\ of\ rows\ first\ input, number\ of\ rows\ second\ input) - 1))) + 1$$

  $$ideal\ accumulator\ fraction\ length = ideal\ product\ output\ fraction\ length$$

If both inputs are complex:

$$ideal\ accumulator\ word\ length = ideal\ product\ output\ word\ length$$
$$+ floor(\log 2(\text{MIN}$$
$$((number\ of\ rows\ first\ input, number\ of\ rows\ second\ input) - 1))) + 2$$

$$ideal\ accumulator\ fraction\ length =$$
$$ideal\ product\ output\ fraction\ length$$

Note The actual accumulator word length may be equal to or greater than the calculated ideal product output word length, depending on the settings on the **Hardware Implementation** pane of the Configuration Parameters dialog box.

- When you select `Same as product output`, these characteristics will match those of the product output.

- When you select `Same as first input`, these characteristics will match those of the first input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the accumulator, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the accumulator. This block requires power-of-two slope and a bias of zero.

Output
Choose how you will specify the data type and scaling of the output of the block:

- When you select `Same as accumulator`, these characteristics will match those of the accumulator.

  A special case occurs when `Inherit via internal rule` is specified for **Accumulator**, and both block inputs are complex. In that case, the output word length will be one less than the accumulator word length.

Convolution

- When you select `Same as product output`, these characteristics will match those of the product output.

- When you select `Same as first input`, these characteristics will match those of the first input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the output, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the output. This block requires power-of-two slope and a bias of zero.

Lock scaling against changes by the autoscaling tool
Select this parameter to prevent any fixed-point scaling you specify in this block mask from being overriden by the autoscaling tool in the Fixed-Point Settings interface. For more information about the autoscaling tool, refer to "Fixed-Point Settings Interface" on page 8-28.

Supported Data Types

- Double-precision floating point

- Single-precision floating point

- Fixed point (signed only)

- 8-, 16-, and 32-bit signed integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Correlation	Signal Processing Blockset
conv	MATLAB

Purpose   Compute cross-correlation of two inputs

Library   Statistics

Description

> XCORR

The Correlation block computes the cross-correlation of analogous columns of an M_u-by-N input matrix u and an M_v-by-N input matrix v. This block can also independently cross-correlate a single-channel column vector with each channel of a multiple-channel matrix.

The frame status of both inputs to the Correlation block must be the same. The block does not accept sample-based full-dimension matrix inputs or 2-D row vector inputs. The outputs are always sample based.

The Convolution block accepts both real and complex floating-point and fixed-point inputs. Fixed-point signals are not supported for the frequency domain.

Correlating Frame-Based Inputs

Matrix inputs to the Correlation block must be frame based. The output, y, is a frame-based (M_u+M_v-1)-by-N matrix whose jth column has elements

$$y_{(i+M_v),\,j} = \sum_{k=1}^{max(M_u,\,M_v)} u_{k,\,j} v^{*}_{(k-i),\,j} \qquad -M_u < i < M_v$$

where * denotes the complex conjugate. Inputs u and v are zero when indexed outside of their valid ranges. When both inputs are real, the output is real; when one or both inputs are complex, the output is complex.

When one input is a column vector (single channel) and the other is a matrix (multiple channels), the single-channel input is independently cross-correlated with each channel of the multichannel input. For example, when u is a M_u-by-1 column vector and v is an M_v-by-N matrix, the output is an (M_u+M_v-1)-by-N matrix whose jth column has elements

Correlation

$$y_{(i+M_v),\, j} = \sum_{k=1}^{max(M_u,\, M_v)} u_k v_{(k-i),\, j}^* \qquad\qquad -M_u < i < M_v$$

Correlating Sample-Based Inputs

The Correlation block does not support sample-based matrix inputs or 2-D row vector inputs. Therefore, all sample-based inputs are column vectors or 1-D vectors. When u and v are sample-based vectors with lengths M_u and M_v, the Correlation block performs the vector cross-correlation

$$y_{(i+M_v)} = \sum_{k=1}^{max(M_u,\, M_v)} u_k v_{(k-i)}^* \qquad\qquad -M_u < i < M_v$$

The dimensions of the sample-based output vector are determined by the dimensions of the input vectors:

- When both inputs are column vectors, or when one input is a column vector and the other is a 1-D vector, the output is a (M_u+M_v-1)-by-1 column vector.

- When both inputs are 1-D vectors, the output is a 1-D vector of length M_u+M_v-1.

Fixed-Point Data Types

The following diagram shows the data types used within the Correlation block for fixed-point signals (time domain only).
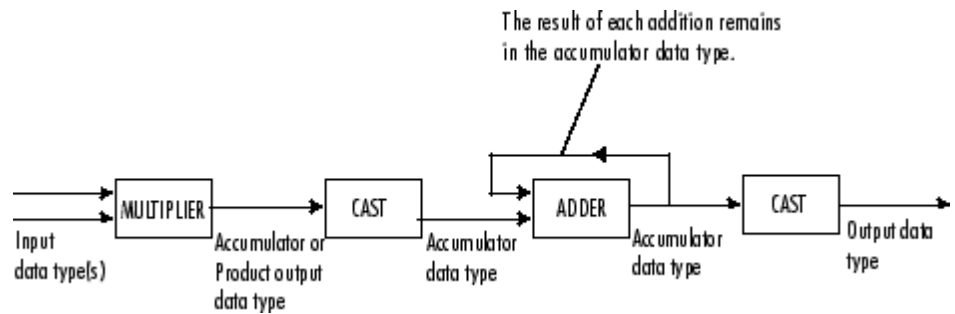
The result of each addition remains
in the accumulator data type.

```
Input                    Accumulator or      Accumulator       Accumulator      Output data
data type(s)             Product output      data type         data type        type
                         data type
```

MULTIPLIER → CAST → ADDER → CAST

You can set the product output, accumulator, and output data types in the block dialog as discussed below.

The output of the multiplier is in the product output data type when the input is real. When the input is complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed, refer to "Multiplication Data Types" on page 8-16.

Correlation

Dialog Box

The **Main** pane of the Correlation block dialog appears as follows:
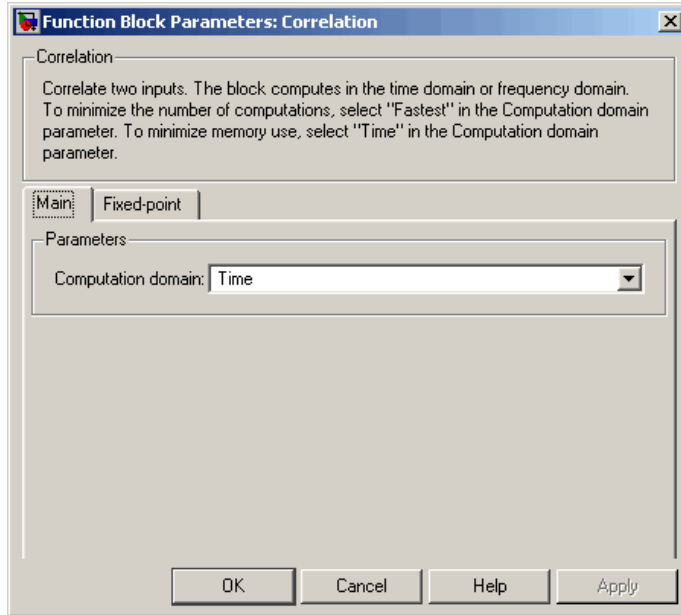
Computation domain

Set the domain in which the block computes correlations:

- Time — The block computes in the time domain, which minimizes memory use.

- Frequency — The block computes in the frequency domain, which might require fewer computations than computing in the time domain, depending on the input length.

- Fastest — The block computes in the domain, which minimizes the number of computations.

The **Fixed-point** pane of the Correlation block dialog appears as follows:
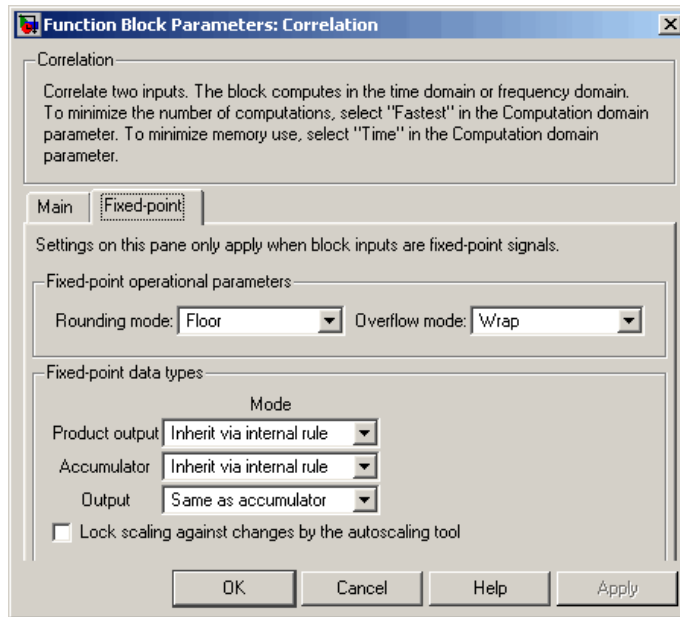
Note Fixed-point signals are only supported for the time domain. To use the parameters on this pane, make sure Time is selected for the **Computation domain** parameter on the **Main** pane.

Rounding mode
    Select the rounding mode for fixed-point operations.

Overflow mode
    Select the overflow mode for fixed-point operations.

Product output
    Use this parameter to specify how you would like to designate the product output word and fraction lengths. Refer to "Fixed-Point

Correlation

Data Types" on page 10-8 and "Multiplication Data Types" on page 8-16 for illustrations depicting the use of the product output data type in this block:

- When you select `Inherit via internal rule`, the product output word length and fraction length are automatically set according to the following equations:

$$ideal\ product\ output\ word\ length\ =\ word\ length\ of\ first\ input\ +\ word\ length\ of\ second\ input$$

$$ideal\ product\ output\ fraction\ length\ =\ fraction\ length\ of\ first\ input\ +\ fraction\ length\ of\ second\ input$$

Note The actual product output word length may be equal to or greater than the calculated ideal product output word length, depending on the settings on the **Hardware Implementation** pane of the Configuration Parameters dialog box.

- When you select `Same as first input`, these characteristics will match those of the first input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the product output, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the product output. This block requires power-of-two slope and a bias of zero.

Accumulator

As depicted above, inputs to the accumulator are cast to the accumulator data type. The output of the adder remains in the accumulator data type as each element of the input is added to it. Use this parameter to specify how you would like to designate this accumulator word and fraction lengths.

You also use this parameter to specify the accumulator word and fraction lengths resulting from a complex-complex multiplication in the block. Refer to "Multiplication Data Types" on page 8-16 for more information.

- When you select Inherit via internal rule, the accumulator word length and fraction length are automatically set according to the following equations:

    If at least one of the inputs is real:

$$ideal\ accumulator\ word\ length = ideal\ product\ output\ word\ length \\ + \text{floor}(\log 2(\text{MIN} \\ ((number\ of\ rows\ first\ input, number\ of\ rows\ second\ input) - 1))) + 1$$

$$ideal\ accumulator\ fraction\ length = \\ ideal\ product\ output\ fraction\ length$$

If both inputs are complex:

$$ideal\ accumulator\ word\ length\ =\ ideal\ product\ output\ word\ length$$
$$+\ \mathrm{floor}(\log 2(\mathrm{MIN}$$
$$((number\ of\ rows\ first\ input,\ number\ of\ rows\ second\ input) - 1))) + 2$$

$$ideal\ accumulator\ fraction\ length =$$
$$ideal\ product\ output\ fraction\ length$$

> **Note** The actual accumulator word length may be equal to or greater than the calculated ideal product output word length, depending on the settings on the **Hardware Implementation** pane of the Configuration Parameters dialog box.

- When you select Same as product output, these characteristics will match those of the product output.

- When you select Same as first input, these characteristics will match those of the first input to the block.

- When you select Binary point scaling, you are able to enter the word length and the fraction length of the accumulator, in bits.

- When you select Slope and bias scaling, you are able to enter the word length, in bits, and the slope of the accumulator. This block requires power-of-two slope and a bias of zero.

Output

Choose how you will specify the word length and fraction length of the output of the block:

- When you select Same as accumulator, these characteristics will match those of the accumulator.

  A special case occurs when Inherit via internal rule is specified for **Accumulator**, and both block inputs are complex.

In that case, the output word length will be one less than the accumulator word length.

- When you select Same as product output, these characteristics will match those of the product output.

- When you select Same as first input, these characteristics will match those of the first input to the block.

- When you select Binary point scaling, you are able to enter the word length and the fraction length of the output, in bits.

- When you select Slope and bias scaling, you are able to enter the word length, in bits, and the slope of the output. This block requires power-of-two slope and a bias of zero.

Lock scaling against changes by the autoscaling tool
Select this parameter to prevent any fixed-point scaling you specify in this block mask from being overriden by the autoscaling tool in the Fixed-Point Settings interface. For more information about the autoscaling tool, refer to "Fixed-Point Settings Interface" on page 8-28.

Supported Data Types
- Double-precision floating point

- Single-precision floating point

- Fixed point (signed only)

- 8-, 16-, and 32-bit signed integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

Correlation

See Also

Autocorrelation	Signal Processing Blockset
Convolution	Signal Processing Blockset
xcorr	Signal Processing Toolbox

Purpose　　　　Count up or down through specified range of numbers

Library　　　　Signal Management / Switches and Counters
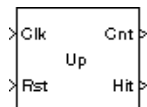
Description　　The Counter block increments or decrements an internal counter each
time it receives a trigger event at the Clk port. A trigger event at the
Rst port resets the counter to its initial state.

The input to the Rst port must be a real sample based scalar. The input
to the Clk port can be a real sample-based scalar, or a real frame-based
vector (that is, single channel). When both inputs are sample based,
they must have the same sample period. When the Clk input is frame
based, the frame period must equal the sample period of the Rst input.

Sections of This Reference Page

- "Setting the Count Event Parameter" on page 10-147

- "Setting the Counter Size and Initial Count Parameters" on page
  10-149

- "Sample-Based Operation" on page 10-150

- "Frame-Based Operation" on page 10-151

- "Free-Running Operation" on page 10-151

- "Examples" on page 10-152

- "Dialog Box" on page 10-155

- "Supported Data Types" on page 10-157

- "See Also" on page 10-158

Setting the Count Event Parameter

The trigger event for both inputs is specified by the **Count event**
parameter, and can be one of the following:

- Rising edge — Triggers a count or reset operation when the Clk or
  Rst input does one of the following:

- Rises from a negative value to a positive value or zero

- Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)
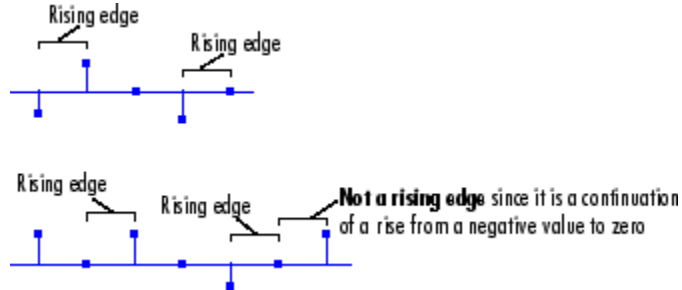
- **Falling edge** — Triggers a count or reset operation when the `Clk` or `Rst` input does one of the following:

  - Falls from a positive value to a negative value or zero

  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero (see the following figure)

- **Either edge** — Triggers a count or reset operation when the `Clk` or `Rst` input is a `Rising edge` or `Falling edge` (as described above).

- Non-zero sample — Triggers a count or reset operation at each sample time when the Clk or Rst input is not zero.

- Free running disables the Clk port, and enables the **Samples per output frame** and **Sample time** parameters. The block increments or decrements the counter at a constant interval, T_s, specified by the **Sample time** parameter (for more information, see "Free-Running Operation" on page 10-151). The Rst port behaves as if the **Count event** parameter were set to Non-zero sample.

Note When running simulations in the Simulink MultiTasking mode, sample-based reset signals have a one-sample latency, and frame-based reset and clock signals have one frame of latency. Thus, there is a one-sample or one-frame delay between the time the block detects a trigger event at the Clk or Rst port, and when it applies the trigger. For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" on page 2-56 and "Models with Multiple Sample Rates" in the Real-Time Workshop User's Guide documentation.

When running simulations in the Simulink MultiTasking mode, sample-based reset signals have a one-sample latency, and frame-based reset signals have one frame of latency. Thus, there is a one-sample or one-frame delay between the time the block detects a reset event, and when it applies the reset. For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" on page 2-56 and "Models with Multiple Sample Rates" in the Real-Time Workshop User's Guide documentation.

Setting the Counter Size and Initial Count Parameters

At the start of the simulation, the block sets the counter to the value specified by the **Initial count** parameter, which can be any integer in the range defined by the **Counter size** parameter. The **Counter size** parameter allows you to choose from three standard counter ranges, or to specify an arbitrary counter limit:

- `8 bits` specifies a counter with a range of 0 to 255.

- `16 bits` specifies a counter with a range of 0 to 65535.

- `32 bits` specifies a counter with a range of 0 to 2^{32}-1.

- `User defined` enables the supplementary **Maximum count** parameter, which allows you to specify an arbitrary integer as the upper count limit. The range of the counter is then 0 to the **Maximum count** value.

Sample-Based Operation

The block operates in sample-based mode when the `Clk` input is a sample-based scalar. Sample-based vectors and matrices are not accepted.

When the **Count direction** parameter is set to `Up`, a sample-based trigger event at the `Clk` input causes the block to increment the counter by one. The block continues incrementing the counter when triggered until the counter value reaches the upper count limit (that is 255 for an 8-bit counter). At the next `Clk` trigger event, the block resets the counter to 0, and resumes incrementing the counter with the subsequent `Clk` trigger event.

When the **Count direction** parameter is set to `Down`, a sample-based trigger event at the `Clk` input causes the block to decrement the counter by one. The block continues decrementing the counter when triggered until the counter value reaches 0. At the next `Clk` trigger event, the block resets the counter to the upper count limit (that is 255 for an 8-bit counter), and resumes decrementing the counter with the subsequent `Clk` trigger event.

Between triggering events the block holds the output at its most recent value. The block resets the counter to its initial state when the trigger event specified in the **Count event** menu is received at the optional `Rst` input. When trigger events are received simultaneously at the `Clk` and `Rst` ports, the block first resets the counter, and then increments or decrements appropriately. (If you do not need to reset the counter

during the simulation, you can disable the Rst port by clearing the **Reset input** check box.)

The **Output** pop-up menu provides three options for the output port configuration of the block icon:

- Count configures the block icon to show a Cnt port, which produces the current value of the counter as a sample-based scalar with the same sample period as the inputs.

- Hit configures the block icon to show a Hit port. The Hit port produces zeros while the value of the counter does not equal the integer **Hit value** parameter setting. When the counter value *does* equal the **Hit value** setting, the block generates a value of 1 at the Hit port. The output is sample based with the same sample period as the inputs.

- Count and Hit configures the block icon with both ports.

Frame-Based Operation

The block operates in frame-based mode when the Clk input is a frame-based vector (that is, single channel). Multichannel frame-based inputs are not accepted.

Frame-based operation is the same as sample-based operation, except that the block increments or decrements the counter by the total number of trigger events contained in the Clk input frame. A trigger event that is split across two consecutive frames is counted in the frame that contains the conclusion of the event. When a trigger event is received at the Rst port, the block first resets the counter, and then increments or decrements the counter by the number of trigger events contained in the Clk frame.

The Cnt and Hit outputs are sample-based scalars with sample period equal to the Clk input frame period.

Free-Running Operation

The block operates in free-running mode when you select Free running from the **Count event** menu.

Counter

The Rst port behaves as if the **Count event** parameter were set to Non-zero sample (triggers a reset at each sample time that the Rst input is not zero).

The Clk input port is disabled in this mode, and the block simply increments or decrements the counter using the constant sample period specified by the **Sample time** parameter, T_s. The Cnt output is a frame-based M-by-1 matrix containing the count value at each of M consecutive sample times, where M is specified by the **Samples per output frame** parameter. The Hit output is a frame-based M-by-1 matrix containing the hit status (0 or 1) at each of those M consecutive sample times. Both outputs have a frame period of $M*T_s$.

Examples In the model below, the Clk port of the Counter block is driven by the Simulink Pulse Generator block, and the Rst port is triggered by an N-Sample Enable block. All of the Counter block's inputs and outputs are multiplexed into a single To Workspace block using a 4-port Mux block.
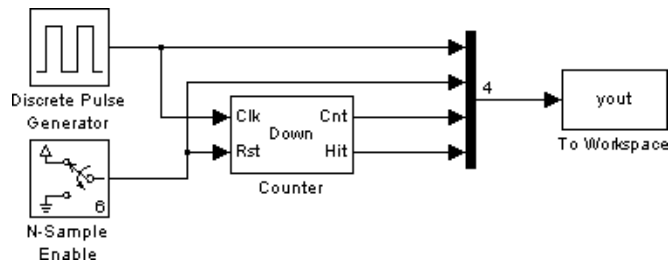
To run the model, first select Configuration Parameters from the **Simulation** menu. In the **Select** pane, click **Solver**, and set the **Stop time** to 30. Then adjust the block parameters as described below. (Use the default settings for the Pulse Generator and To Workspace blocks.)

- Set the N-Sample Enable block parameters as follows:

  - **Trigger count** = 6
  - **Active level** = High (1)

- Set the Counter block parameters as follows:
  - **Count direction:** Down
  - **Count event:** Rising edge
  - **Counter size:** User defined
  - **Maximum count:** 20
  - **Initial count:** 5
  - **Output:** Count and Hit
  - **Hit value:** 4
  - **Reset input**

    ☑

  - **Count data type:** Double
  - **Hit data type:** Logical
- Set the **Number of inputs** parameter of the Mux block to 4.

The figure below shows the first 22 samples of the model's four-column output, yout. The first column is the Counter block's Clk input, the second column is the block's Rst input, the third column is the block's Cnt output, and the fourth column is the block's Hit output.

Clk	Rst		Cnt	Hit
[1]	[0]	initial value	[5]	[0]
[0]	[0]		[5]	[0]
[1]	[0]		[4]	hit [1]
[0]	[0]		[4]	[1]
[1]	[0]		[3]	[0]
[0]	[0]		[3]	[0]
[1]	[1]	reset	[4]	hit [1]
[0]	[1]		[4]	[1]
[1]	[1]		[3]	[0]
[0]	[1]		[3]	[0]
[1]	[1]		[2]	[0]
[0]	[1]		[2]	[0]
[1]	[1]		[1]	[0]
[0]	[1]		[1]	[0]
[1]	[1]		[0]	[0]
[0]	[1]		[0]	[0]
[1]	[1]		[20]	[0]
[0]	[1]	max value	[20]	[0]
[1]	[1]		[19]	[0]
[0]	[1]		[19]	[0]
[1]	[1]		[18]	[0]
[0]	[1]		[18]	[0]

Simulation time

Clk   Cnt
Up
Rst   Hit
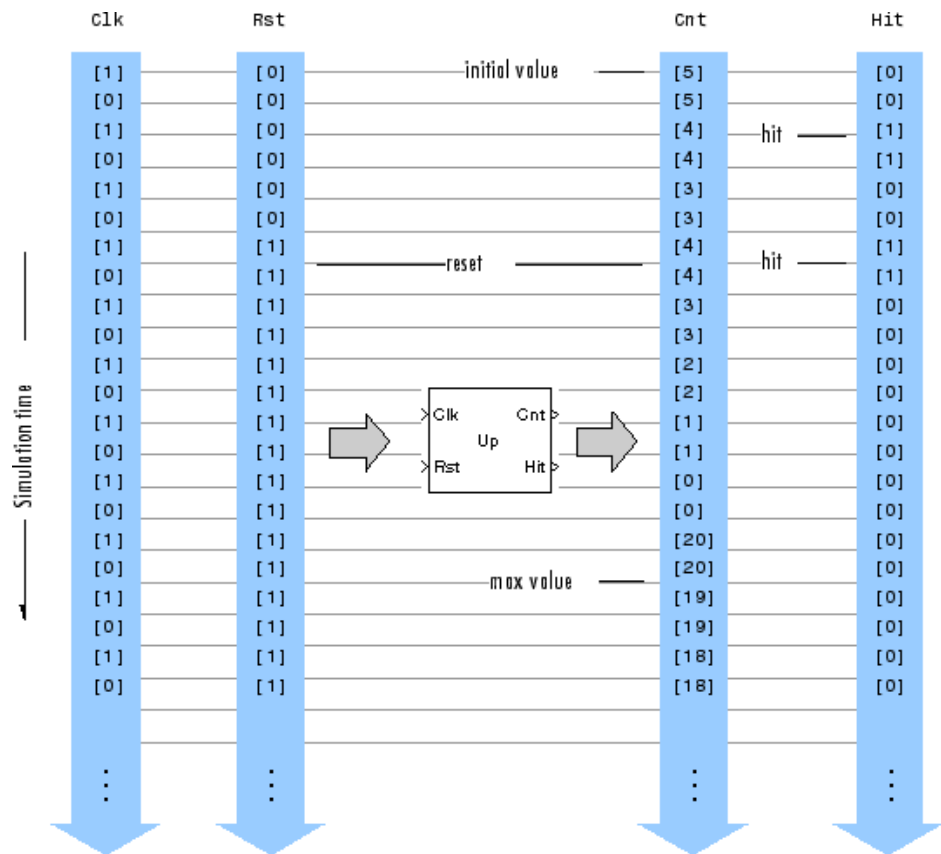
You can see that the seventh input samples to both the Clk and Rst ports of the Counter block represent trigger events (rising edges), so at this time step the block first resets the counter to its initial value of 5, and then immediately decrements the count to 4. When the counter reaches its minimum value of 0, it rolls over to its maximum value of 20 with the following trigger event at the Cnt port.

Dialog Box

Count direction

The counter direction, Up or Down. Tunable, except in the Simulink external mode.

Count event

The type of event that triggers the block to increment, decrement, or reset the counter when received at the Clk or Rst ports. Free running disables the Clk port, and counts continuously with

the period specified by the **Sample time** parameter. For more information on all the possible settings, see "Setting the Count Event Parameter" on page 10-147.

Counter size

The range of integer values the block should count through before recycling to zero. For more information, see "Setting the Counter Size and Initial Count Parameters" on page 10-149.

Maximum count

The counter's maximum value when **Counter size** is set to User defined. Tunable.

Initial count

The counter's initial value at the start of the simulation and after reset. Tunable, except in the Simulink external mode.

Output

Selects the output port(s) to enable: Cnt, Hit, or both.

Hit value

The scalar value whose occurrence in the count should be flagged by a 1 at the (optional) Hit output. This parameter is available when Hit or Count and Hit are selected in the **Output** menu. Tunable, except in the Simulink external mode.

Reset input

Enables the Rst input port when selected.

Samples per output frame

The number of samples, *M*, in each output frame. This parameter is available when you select Free running in the **Count event** menu.

Sample time

The output sample period, T_s, in free-running mode. This parameter is available when you select Free running in the **Count event** menu.

Count data type

> The data type of the output from the Cnt output port. This parameter is available when the **Output** parameter is set to Count or Count and Hit.

Hit data type

> The data type of the output from the Hit output port. For information on the Logical and Boolean options of this parameter, see "Effects of Enabling and Disabling Boolean Support" on page 7-17. This parameter is available when the **Output** parameter is set to Hit or the **Output** parameter is set to Count and Hit and the **Count data type** parameter is set to Double.

Supported Data Types

Port	Supported Data Types
Clk	• Double-precision floating point
	• Single-precision floating point
	• Boolean
Rst	• Double-precision floating point
	• Single-precision floating point
	• Boolean

Counter

Port	Supported Data Types
Cnt	• Double-precision floating point • Single-precision floating point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Hit	• Logical • Boolean — The block might output Boolean values from the Hit output port depending on the **Hit data type** parameter setting, as described in "Effects of Enabling and Disabling Boolean Support" on page 7-17. To learn how to disable Boolean output support, see "Steps to Disabling Boolean Support" on page 7-18.

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Edge Detector	Signal Processing Blockset
N-Sample Enable	Signal Processing Blockset
N-Sample Switch	Signal Processing Blockset

Purpose   Compute estimate of autoregressive (AR) model parameters using covariance method

Library   Estimation / Parametric Estimation

Description   The Covariance AR Estimator block uses the covariance method to fit an autoregressive (AR) model to the input data. This method minimizes the forward prediction error in the least squares sense.

The input is a sample-based vector (row, column, or 1-D) or frame-based vector (column only) representing a frame of consecutive time samples from a single-channel signal, which is assumed to be the output of an AR system driven by white noise. The block computes the normalized estimate of the AR system parameters, $A(z)$, independently for each successive input frame.

$$H(z) = \frac{G}{A(z)} = \frac{G}{1 + a(2)z^{-1} + \ldots + a(p+1)z^{-p}}$$

The order, p, of the all-pole model is specified by the **Estimation order** parameter. To guarantee a valid output, you must set the **Estimation order** parameter to be less than or equal to half the input vector length.

The top output, A, is a column vector of length $p+1$ with the same frame status as the input, and contains the normalized estimate of the AR model coefficients in descending powers of z.

```
[1 a(2) ... a(p+1)]
```

The scalar gain, G, is provided at the bottom output (G).

See the Burg AR Estimator block reference page for a comparison of the Burg AR Estimator, Covariance AR Estimator, Modified Covariance AR Estimator, and Yule-Walker AR Estimator blocks.

Covariance AR Estimator

Dialog Box
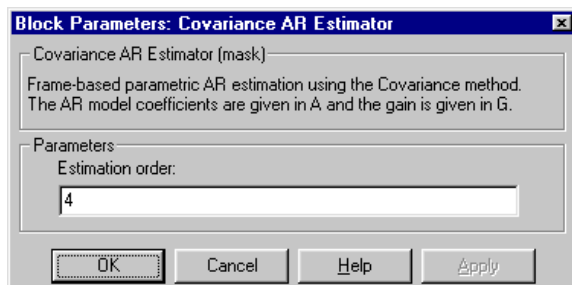
Estimation order

The order of the AR model, *p*. To guarantee a nonsingular output, you must set *p* to be less than the input length. Otherwise, the output might be singular.

References Kay, S. M. *Modern Spectral Estimation: Theory and Application.* Englewood Cliffs, NJ: Prentice-Hall, 1988.

Marple, S. L., Jr., *Digital Spectral Analysis with Applications.* Englewood Cliffs, NJ: Prentice-Hall, 1987.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
A	• Double-precision floating point
	• Single-precision floating point
G	• Double-precision floating point
	• Single-precision floating point

The output data type is the same as the input data type. To learn how to convert your data types to the above data types in MATLAB and

Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Burg AR Estimator	Signal Processing Blockset
Covariance Method	Signal Processing Blockset
Modified Covariance AR Estimator	Signal Processing Blockset
Yule-Walker AR Estimator	Signal Processing Blockset
`arcov`	Signal Processing Toolbox

Covariance Method

Purpose

Compute parametric spectral estimate using covariance method

Library

Estimation / Power Spectrum Estimation

Description
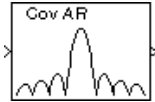
Cov AR

The Covariance Method block estimates the power spectral density (PSD) of the input using the covariance method. This method fits an autoregressive (AR) model to the signal by minimizing the forward prediction error in the least squares sense. The order of the all-pole model is the value specified by the **Estimation order** parameter, and the spectrum is computed from the FFT of the estimated AR model parameters. To guarantee a valid output, you must set the **Estimation order** parameter to be less than or equal to half the input vector length.

The input is a sample-based vector (row, column, or 1-D) or frame-based vector (column only) representing a frame of consecutive time samples from a single-channel signal. The block's output (a column vector) is the estimate of the signal's power spectral density at N_{fft} equally spaced frequency points in the range $[0, F_s)$, where F_s is the signal's sample frequency.

When you select **Inherit FFT length from input dimensions**, N_{fft} is specified by the frame size of the input, which must be a power of 2. When you do *not* select **Inherit FFT length from input dimensions**, N_{fft} is specified as a power of 2 by the **FFT length** parameter, and the block zero pads or truncates the input to N_{fft} before computing the FFT. The output is always sample based.

See the Burg Method block reference for a comparison of the Burg Method, Covariance Method, Modified Covariance Method, and Yule-Walker Method blocks.

Dialog Box
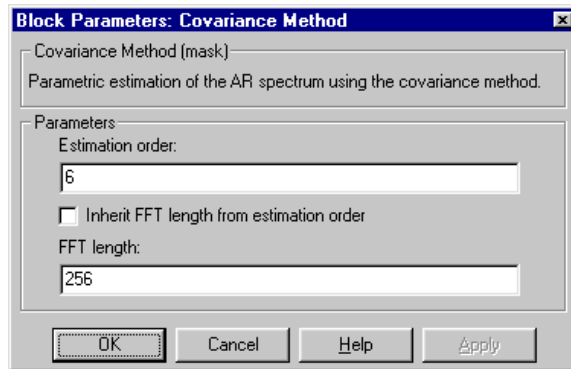
Estimation order

The order of the AR model. To guarantee a nonsingular output, you must set the value of this parameter to be less than the input length. Otherwise, the output might be singular.

Inherit FFT length from input dimensions

When selected, uses the input frame size as the number of data points, N_{fft}, on which to perform the FFT. Tunable.

FFT length

The number of data points, N_{fft}, on which to perform the FFT. When N_{fft} exceeds the input frame size, the frame is zero-padded as needed. This parameter is enabled when you do not select **Inherit FFT length from input dimensions**.

References
Kay, S. M. *Modern Spectral Estimation: Theory and Application.* Englewood Cliffs, NJ: Prentice-Hall, 1988.

Marple, S. L., Jr., *Digital Spectral Analysis with Applications.* Englewood Cliffs, NJ: Prentice-Hall, 1987.

Covariance Method

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point

The output data type is the same as the input data type. To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Burg Method	Signal Processing Blockset
Covariance AR Estimator	Signal Processing Blockset
Modified Covariance Method	Signal Processing Blockset
Short-Time FFT	Signal Processing Blockset
Yule-Walker Method	Signal Processing Blockset
pcov	Signal Processing Toolbox

See "Power Spectrum Estimation" on page 6-6 for related information.

Purpose   Create square diagonal matrix from diagonal elements

Library   Math Functions / Matrices and Linear Algebra / Matrix Operations
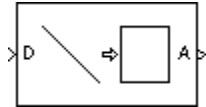
Description   The Create Diagonal Matrix block populates the diagonal of the M-by-M matrix output with the elements contained in the length-M vector input, D. The elements off the diagonal are zero.

```
A = diag(D)      Equivalent MATLAB code
```

The output is always sample based.

Dialog Box

Block Parameters: Create Diagonal Matrix
Create Diagonal Matrix (mask) (link)
Convert a vector of diagonal elements to a full matrix.

OK   Cancel   Help   Apply
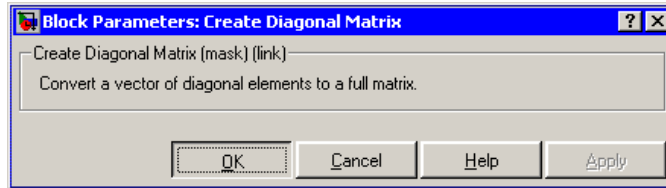
Create Diagonal Matrix

Supported Data Types

Port	Supported Data Types
D	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Custom data types
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
A	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Custom data types
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Constant Diagonal Matrix	Signal Processing Blockset
Extract Diagonal	Signal Processing Blockset
diag	MATLAB

Purpose   Compute cumulative product of channel, column, or row elements

Library   Math Functions / Math Operations

Description   The Cumulative Product block computes the cumulative product of elements in each channel, column, or row of the *M*-by-*N* input matrix.

The inputs can be sample-based or frame-based vectors and matrices. The output always has the same dimensions, rate, frame status, data type, and complexity as the input.

The Cumulative Product block accepts real and complex fixed-point and floating-point inputs.

Sections of This Reference Page

- "Input and Output Characteristics" on page 10-167

- "Multiplying Along Channels" on page 10-168

- "Resetting the Cumulative Product Along Channels" on page 10-170

- "Multiplying Along Columns" on page 10-172

- "Multiplying Along Rows" on page 10-173

- "Dialog Box" on page 10-175

- "Supported Data Types" on page 10-179

- "See Also" on page 10-180

Input and Output Characteristics

Valid Input

The block computes the cumulative product of both sample- and frame-based vector and matrix inputs. Inputs can be real or complex. When multiplying along channels or columns, 1-D unoriented vectors are treated as column vectors. When multiplying along rows, 1-D vectors are treated as row vectors.

Cumulative Product

Valid Reset Signal

The optional reset port, `Rst`, accepts scalar values, which can be any built-in Simulink data type including `boolean`. The rate of the reset signal must be a positive integer multiple of the rate of the data signal input.

Output Characteristics

The output always has the same dimensions, rate, frame status, data type, and complexity as the data signal input.

Multiplying Along Channels

When the **Multiply input along** parameter is set to `Channels (running product)`, the block computes the cumulative product of the elements in each input channel. The running product of the current input takes into account the running product of all previous inputs. See the following sections for more information:

- "Multiplying Along Channels of Frame-Based Inputs" on page 10-168
- "Multiplying Along Channels of Sample-Based Inputs" on page 10-169
- "Resetting the Cumulative Product Along Channels" on page 10-170

Multiplying Along Channels of Frame-Based Inputs

For frame-based inputs, the block treats each input column as an independent channel. As the following figure and equation illustrate, the output has the following characteristics:

- The first row of the first output is the same as the first row of the first input.
- The first row of each subsequent output is the element-wise product of the first row of the current input (time t), and the last row of the previous output (time t - T_f, where T_f is the frame period).
- The output has the same size, dimension, frame status, data type, and complexity as the input.

Given an M-by-N frame-based input, u, the output, y, is a frame-based M-by-N matrix whose first row has elements

$$y_{1,j}(t) = u_{1,j}(t) \cdot y_{M,j}(t - T_f)$$

Product Along Channels for Frame-Based Inputs

Multiplying Along Channels of Sample-Based Inputs

For sample-based inputs, the block treats each element of the input matrix as an independent channel. As the following figure and equation illustrate, the output has the following characteristics:

- The first output is the same as the first input.

- Each subsequent output is the element-wise product of the current input (time t) and the previous output (time t - T_s, where T_s is the sample period).

- The output has the same size, dimension, frame status, data type, and complexity as the input.

Given an M-by-N sample-based input, u, the output, y, is a sample-based M-by-N matrix with the elements

$$y_{i,j}(t) = u_{i,j}(t) \cdot y_{i,j}(t - T_s) \qquad \begin{array}{l} 1 \leq i \leq M \\ 1 \leq j \leq N \end{array}$$

Cumulative Product

For convenience, length-*M* 1-D vector inputs are treated as *M*-by-1 column vectors when multiplying along channels, and the output is a length-*M* 1-D vector.

Product Along Channels for Sample-Based Inputs
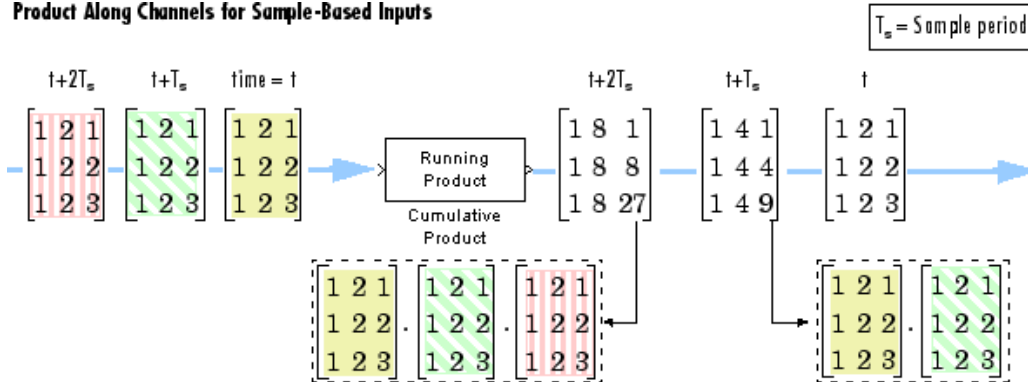
Resetting the Cumulative Product Along Channels

When you set the **Multiply input along** parameter to `Channels (running product)`, you can set the block to reset the running product whenever it detects a reset event at the optional `Rst` port. The rate of the reset signal must be a positive integer multiple of the rate of the data signal input. The input to the `Rst` port can be of the Boolean data type.

When the block is reset for sample-based inputs, the block initializes the current output to the values of the current input. For frame-based inputs, the block initializes the first row of the current output to the values in the first row of the current input.

The **Reset port** parameter specifies the reset event, which can be one of the following:

- `None` disables the `Rst` port.

- `Rising edge` — Triggers a reset operation when the `Rst` input does one of the following:

  - Rises from a negative value to a positive value or zero

- Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)

- Falling edge — Triggers a reset operation when the Rst input does one of the following:

  - Falls from a positive value to a negative value or zero

  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero (see the following figure)

- Either edge — Triggers a reset operation when the Rst input is a Rising edge or Falling edge (as described above)

- Non-zero sample — Triggers a reset operation at each sample time that the Rst input is not zero

Cumulative Product

Multiplying Along Columns

When the **Multiply input along** parameter is set to `Columns`, the block computes the cumulative product of each column of the input, where the current cumulative product is independent of the cumulative products of previous inputs.

```
y = cumprod(u)       % Equivalent MATLAB code
```

The output has the same size, dimension, frame status, data type, and complexity as the input. The mth output row is the element-wise product of the first m input rows.

Given an M-by-N input, u, the output, y, is an M-by-N matrix whose jth column has elements

$$y_{i,j} = \prod_{k=1}^{i} u_{k,j} \qquad 1 \le i \le M$$

The block treats length-M 1-D vector inputs as M-by-1 column vectors when multiplying along columns.

Product Along Columns
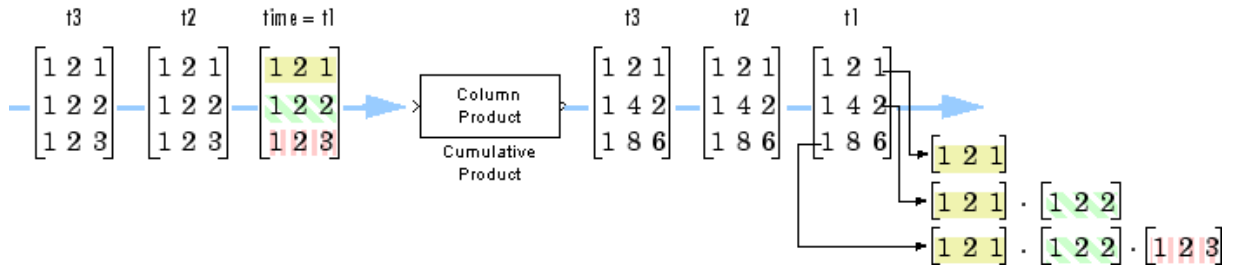
Multiplying Along Rows

When the **Multiply input along** parameter is set to Rows, the block computes the cumulative product of the row elements, where the current cumulative product is independent of the cumulative products of previous inputs.

```
y = cumprod(u,2)      % Equivalent MATLAB code
```

The output has the same size, dimension, frame status, and data type as the input. The *n*th output column is the element-wise product of the first *n* input columns.

Given an *M*-by-*N* input, *u*, the output, *y*, is an *M*-by-*N* matrix whose *i*th row has elements

$$y_{i,j} = \prod_{k=1}^{j} u_{i,k} \qquad 1 \le j \le N$$

The block treats length-*N* 1-D vector inputs as 1-by-*N* row vectors when multiplying along rows.

Cumulative Product

Product Along Rows
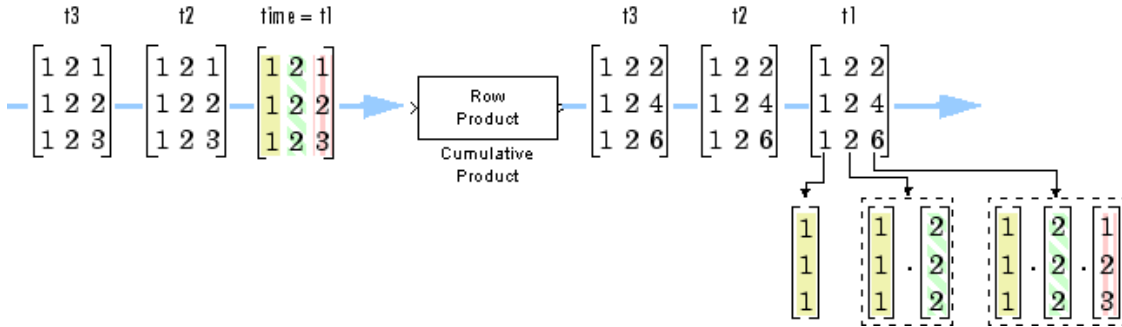
Fixed-Point Data Types

The following diagram shows the data types used within the Cumulative Product block for fixed-point signals.
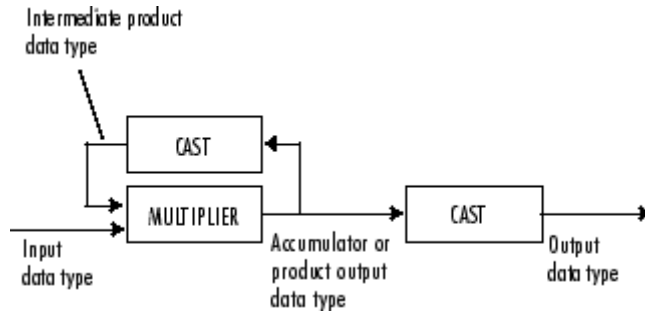
The output of the multiplier is in the product output data type when at least one of the inputs to the multiplier is real. When both of the inputs to the multiplier are complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed, refer to "Multiplication Data Types" on page 8-16. You can set the accumulator, product output, intermediate product, and output data types in the block dialog as discussed in "Dialog Box" on page 10-175.

Dialog Box

The **Main** pane of the Cumulative Product block dialog appears as follows:
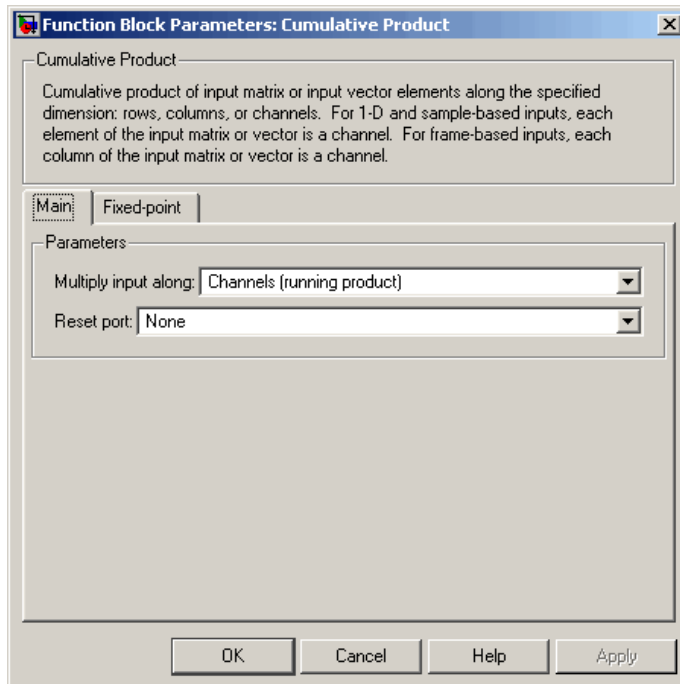
Multiply input along

The dimension along which to compute the cumulative products. The options allow you to multiply along Channels (running product), Columns, and Rows. For more information, see the following sections:

"Multiplying Along Channels" on page 10-168

"Multiplying Along Columns" on page 10-172

"Multiplying Along Rows" on page 10-173

Cumulative Product

Reset port

Determines the reset event that causes the block to reset the product along channels. The rate of the reset signal must be a positive integer multiple of the rate of the data signal input. This parameter is enabled only when you set the **Multiply input along** parameter to Channels (running product). For more information, see "Resetting the Cumulative Product Along Channels" on page 10-170.

The **Fixed-point** pane of the Cumulative Product block dialog appears as follows:
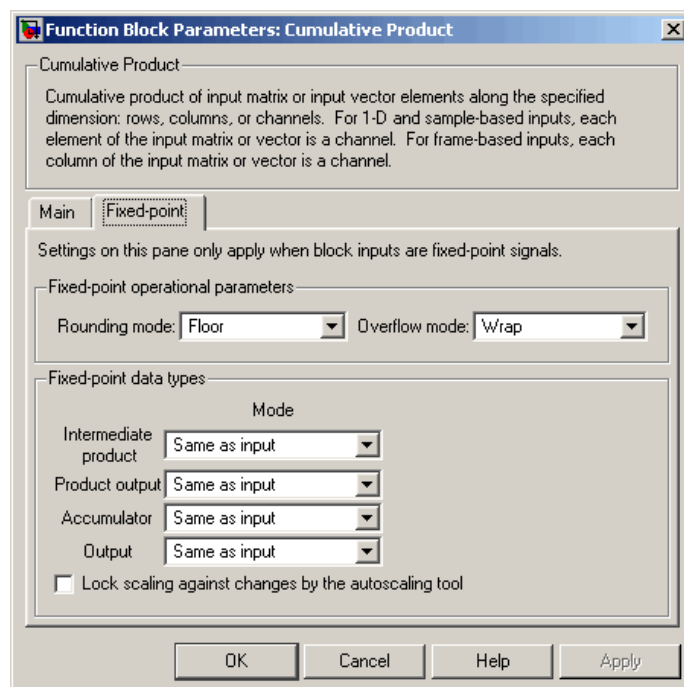
Rounding mode

Select the rounding mode for fixed-point operations.

Overflow mode

Select the overflow mode for fixed-point operations.

Intermediate product

As shown in "Fixed-Point Data Types" on page 10-174, the output of the multiplier is cast to the intermediate product data type before the next element of the input is multiplied into it. Use this parameter to specify how you would like to designate the intermediate product word and fraction lengths:

- When you select Same as input, these characteristics will match those of the input to the block.

- When you select Binary point scaling, you are able to enter the word length and the fraction length of the intermediate product, in bits.

- When you select Slope and bias scaling, you are able to enter the word length, in bits, and the slope of the intermediate product. This block requires power-of-two slope and a bias of zero.

Product output

Use this parameter to specify how you would like to designate the product output word and fraction lengths. Refer to "Fixed-Point Data Types" on page 10-174 and "Multiplication Data Types" on page 8-16 for illustrations depicting the use of the product output data type in this block:

- When you select Same as input, these characteristics will match those of the input to the block.

- When you select Binary point scaling, you are able to enter the word length and the fraction length of the product output, in bits.

- When you select Slope and bias scaling, you are able to enter the word length, in bits, and the slope of the product output. This block requires power-of-two slope and a bias of zero.

Cumulative Product

Accumulator

Use this parameter to specify how you would like to designate the accumulator word and fraction lengths. Refer to "Fixed-Point Data Types" on page 10-174 and "Multiplication Data Types" on page 8-16 for illustrations depicting the use of the accumulator data type in this block. Note that the accumulator data type is only used when both inputs to the multiplier are complex:

- When you select `Same as product output`, these characteristics will match those of the product output.

- When you select `Same as input`, these characteristics will match those of the input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the accumulator, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the accumulator. This block requires power-of-two slope and a bias of zero.

Output

Choose how you will specify the word length and fraction length of the output of the block:

- When you select `Same as product output`, these characteristics will match those of the product output.

- When you select `Same as input`, these characteristics will match those of the input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the output, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the output. This block requires power-of-two slope and a bias of zero.

Lock scaling against changes by the autoscaling tool

Select this parameter to prevent any fixed-point scaling you specify in this block mask from being overriden by the autoscaling

tool in the Fixed-Point Settings interface. For more information about the autoscaling tool, refer to "Fixed-Point Settings Interface" on page 8-28.

Supported Data Types

Input and Output Ports	Supported Data Types
Data input port, `In`	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
Reset input port, `Rst`	All built-in Simulink data types: • Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output port	Always has same data type as data input

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

Cumulative Product

See Also

Cumulative Sum	Signal Processing Blockset
Matrix Product	Signal Processing Blockset
cumprod	MATLAB

Purpose   Compute cumulative sum of channel, column, or row elements

Library   Math Functions / Math Operations

Description   The Cumulative Sum block computes the cumulative sum of the elements in each channel, column, or row of the *M*-by-*N* input matrix.

The inputs can be sample-based or frame-based vectors and matrices. The output always has the same dimensions, rate, frame status, data type, and complexity as the input.

The Cumulative Sum block accepts real and complex fixed-point and floating-point inputs.

Sections of This Reference Page

Input and Output Characteristics

Valid Input

The block computes the cumulative sum of both sample- and frame-based vector and matrix inputs. Inputs can be real or complex. When summing along channels or columns, 1-D unoriented vectors are treated as column vectors. When summing along rows, 1-D vectors are treated as row vectors.

Cumulative Sum

Valid Reset Signal

The optional reset port, Rst, accepts scalar values, which can be any built-in Simulink data type including boolean. The rate of the reset signal must be a positive integer multiple of the rate of the data signal input.

Output Characteristics

The output always has the same dimensions, rate, frame status, data type, and complexity as the data signal input.

Summing Along Channels

When the **Sum input along** parameter is set to Channels (running sum), the block computes the cumulative sum of the elements in each input channel. The running sum of the current input takes into account the running sum of all previous inputs. See the following sections for more information:

- "Summing Along Channels of Frame-Based Inputs" on page 10-182
- "Summing Along Channels of Sample-Based Inputs" on page 10-183
- "Resetting the Cumulative Sum Along Channels" on page 10-184

Summing Along Channels of Frame-Based Inputs

For frame-based inputs, the block treats each input column as an independent channel. As the following figure and equation illustrate, the output has the following characteristics:

- The first row of the first output is the same as the first row of the first input.
- The first row of each subsequent output is the sum of the first row of the current input (time t), and the last row of the previous output (time t - T_f, where T_f is the frame period).
- The output has the same size, dimension, frame status, data type, and complexity as the input.

Given an M-by-N frame-based input, u, the output, y, is a frame-based M-by-N matrix whose first row has elements

$$y_{1,j}(t) = u_{1,j}(t) + y_{M,j}(t - T_f)$$

Sum Along Channels for Frame-Based Inputs
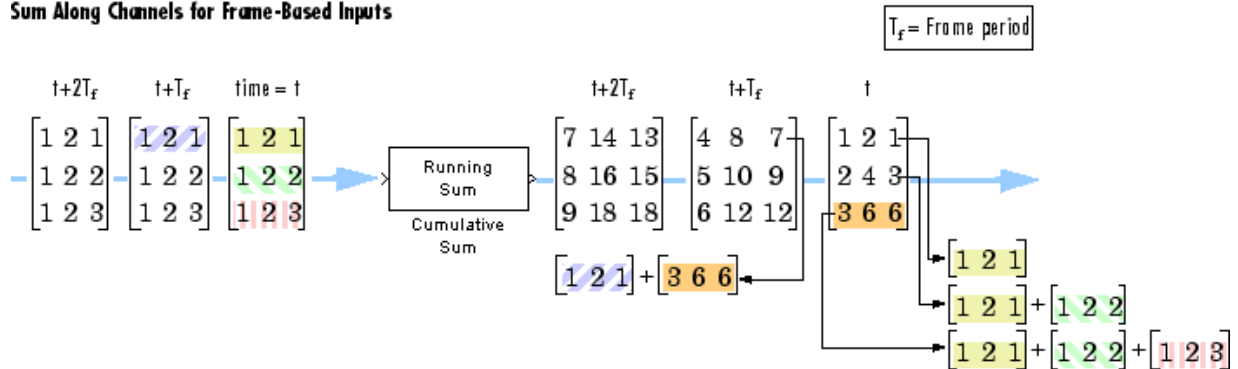
Summing Along Channels of Sample-Based Inputs

For sample-based inputs, the block treats each element of the input matrix as an independent channel. As the following figure and equation illustrate, the output has the following characteristics:

- The first output is the same as the first input.

- Each subsequent output is the sum of the current input (time t) and the previous output (time t - T_s, where T_s is the sample period).

- The output has the same size, dimension, frame status, data type, and complexity as the input.

Given an M-by-N sample-based input, u, the output, y, is a sample-based M-by-N matrix with the elements

$$y_{i,j}(t) = u_{i,j}(t) + y_{i,j}(t - T_s) \qquad \begin{aligned} 1 \le i \le M \\ 1 \le j \le N \end{aligned}$$

Cumulative Sum

Sum Along Channels for Sample-Based Inputs
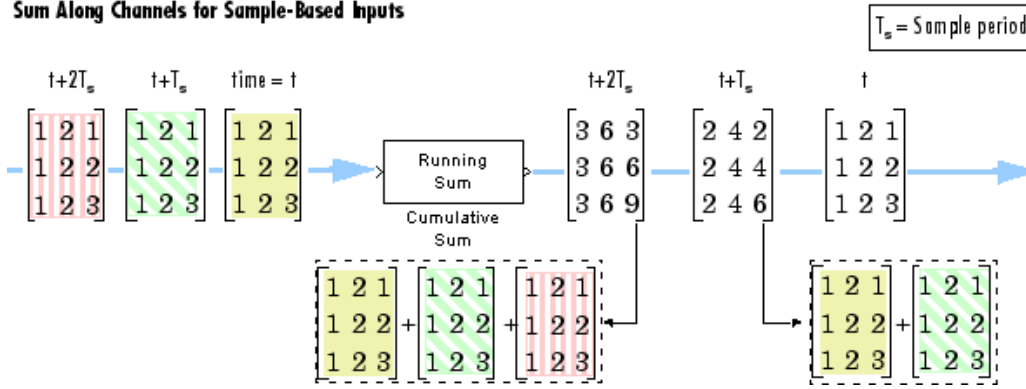
Resetting the Cumulative Sum Along Channels

When you set the **Sum input along** parameter to Channels (running sum), you can set the block to reset the running sum whenever it detects a reset event at the optional Rst port. The rate of the reset signal must be a positive integer multiple of the rate of the data signal input. The input to the Rst port can be of the boolean data type.

When the block is reset for sample-based inputs, the block initializes the current output to the values of the current input. For frame-based inputs, the block initializes the first row of the current output to the values in the first row of the current input.

The **Reset port** parameter specifies the reset event, which can be one of the following:

- None disables the Rst port.

- Rising edge — Triggers a reset operation when the Rst input does one of the following:

  - Rises from a negative value to a positive value or zero

  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)

- Falling edge — Triggers a reset operation when the Rst input does one of the following:

  - Falls from a positive value to a negative value or zero

  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero (see the following figure)

- Either edge — Triggers a reset operation when the Rst input is a Rising edge or Falling edge (as described above)

- Non-zero sample — Triggers a reset operation at each sample time that the Rst input is not zero

Cumulative Sum

Summing Along Columns

When the **Sum input along** parameter is set to `Columns`, the block computes the cumulative sum of each column of the input, where the current cumulative sum is independent of the cumulative sums of previous inputs.

```
y = cumsum(u)        % Equivalent MATLAB code
```

The output has the same size, dimension, frame status, data type, and complexity as the input. The mth output row is the sum of the first m input rows.

Given an M-by-N input, u, the output, y, is an M-by-N matrix whose jth column has elements

$$y_{i,j} = \sum_{k=1}^{i} u_{k,j} \qquad 1 \le i \le M$$

The block treats length-M 1-D vector inputs as M-by-1 column vectors when summing along columns.

Sum Along Columns
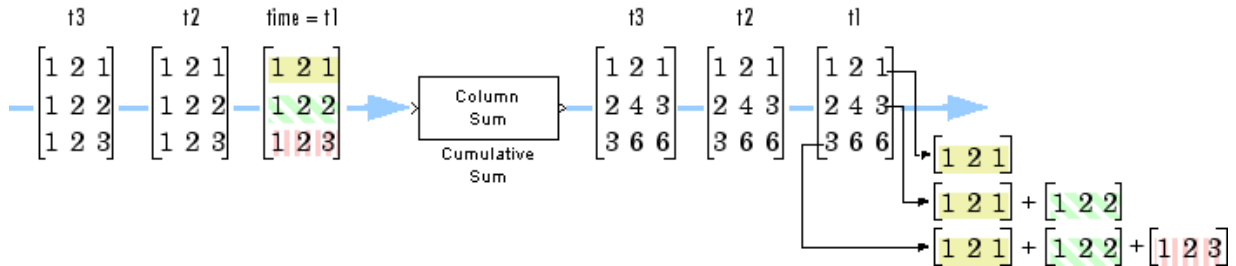
Summing Along Rows

When the **Sum input along** parameter is set to Rows, the block computes the cumulative sum of the row elements, where the current cumulative sum is independent of the cumulative sums of previous inputs.

```
y = cumsum(u,2)       % Equivalent MATLAB code
```

The output has the same size, dimension, frame status, and data type as the input. The nth output column is the sum of the first n input columns.

Given an M-by-N input, u, the output, y, is an M-by-N matrix whose ith row has elements

$$y_{i,j} = \sum_{k=1}^{j} u_{i,k} \qquad 1 \le j \le N$$

The block treats length-N 1-D vector inputs as 1-by-N row vectors when summing along rows.

Cumulative Sum

Sum Along Rows
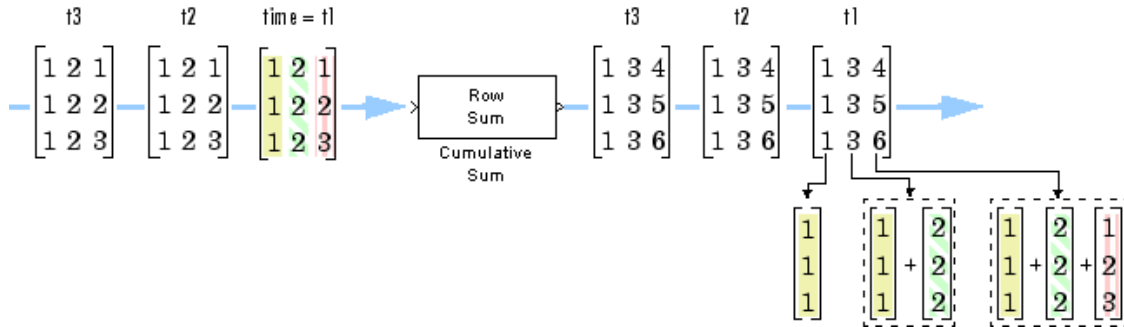
Fixed-Point Data Types

The following diagram shows the data types used within the Cumulative Sum block for fixed-point signals.
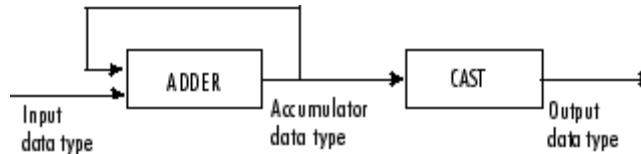
You can set the accumulator and output data types in the block dialog as discussed in "Dialog Box" on page 10-189.

Dialog Box

The **Main** pane of the Cumulative Sum block dialog appears as follows:
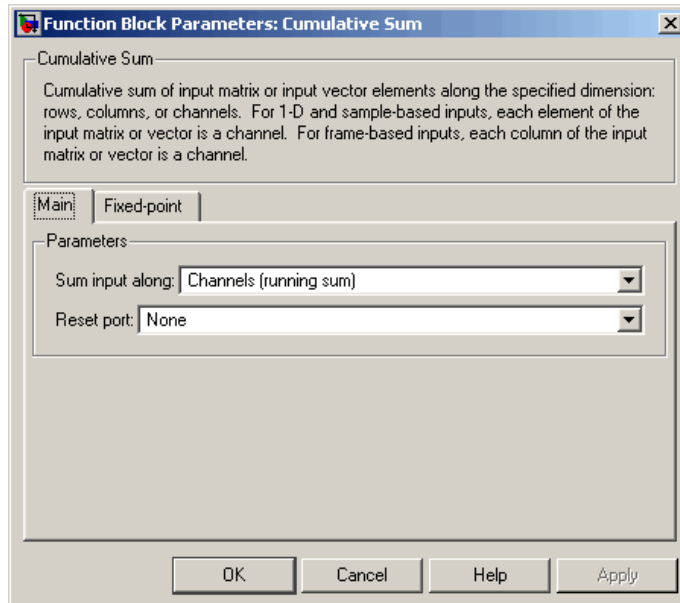
Sum input along

The dimension along which to compute the cumulative summations. The options allow you to sum along Channels (running sum), Columns, and Rows. For more information, see the following sections:

"Summing Along Channels" on page 10-182

"Summing Along Columns" on page 10-186

"Summing Along Rows" on page 10-187

Reset port

Determines the reset event that causes the block to reset the sum along channels. The rate of the reset signal must be a

positive integer multiple of the rate of the data signal input. This parameter is enabled only when you set the **Sum input along** parameter to Channels (running sum). For more information, see "Resetting the Cumulative Sum Along Channels" on page 10-184.

The **Fixed-point** pane of the Cumulative Sum block dialog appears as follows:

Rounding mode

Select the rounding mode for fixed-point operations.

Overflow mode

Select the overflow mode for fixed-point operations.

Accumulator

Use this parameter to specify how you would like to designate this accumulator word and fraction lengths:

- When you select `Same as input`, these characteristics will match those of the input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the accumulator, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the accumulator. This block requires power-of-two slope and a bias of zero.

Output

Choose how you will specify the output word length and fraction length:

- When you select `Same as accumulator`, these characteristics will match those of the accumulator.

- When you select `Same as input`, these characteristics will match those of the input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the output, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the output. This block requires power-of-two slope and a bias of zero.

Lock scaling against changes by the autoscaling tool

Select this parameter to prevent any fixed-point scaling you specify in this block mask from being overriden by the autoscaling tool in the Fixed-Point Settings interface. For more information about the autoscaling tool, refer to "Fixed-Point Settings Interface" on page 8-28.

Cumulative Sum

Supported Data Types

Input and Output Ports	Supported Data Types
Data input port, `In`	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
Reset input port, `Rst`	All built-in Simulink data types: • Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output port	Always has same data type as data input

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Cumulative Product	Signal Processing Blockset
Difference	Signal Processing Blockset

Matrix Sum	Signal Processing Blockset
`cumsum`	MATLAB

dB Conversion

Purpose   Convert magnitude data to decibels (dB or dBm)

Library   Math Functions / Math Operations
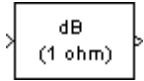
Description   The dB Conversion block converts a linearly scaled power or amplitude input to dB or dBm. The **Input signal** parameter specifies whether the input is a power signal or a voltage signal, and the **Convert to** parameter controls the scaling of the output. When selected, the **Add eps to input to protect against "log(0) = -inf"** parameter adds a value of eps to all power and voltage inputs. When this option is not enabled, zero-valued inputs produce -inf at the output. The size and frame status of the output are the same as the input.

Power Inputs

Select Power as the **Input signal** parameter when the input, u, is a real, nonnegative, power signal (units of watts). When the **Convert to** parameter is set to dB, the block performs the dB conversion

```
y = 10*log10(u)        % Equivalent MATLAB code
```

When the **Convert to** parameter is set to dBm, the block performs the dBm conversion

```
y = 10*log10(u) + 30
```

The dBm conversion is equivalent to performing the dB operation *after* converting the input to milliwatts.

Voltage Inputs

Select Amplitude as the **Input signal** parameter when the input, u, is a real voltage signal (units of volts). The block uses the scale factor specified in ohms by the **Load resistance** parameter, R, to convert the voltage input to units of power (watts) before converting to dB or dBm. When the **Convert to** parameter is set to dB, the block performs the dB conversion

```
y = 10*log10(abs(u)^2/R)
```

When the **Convert to** parameter is set to dBm, the block performs the dBm conversion

```
y = 10*log10(abs(u)^2/R) + 30
```

The dBm conversion is equivalent to performing the dB operation *after* converting the (abs(u)^2/R) result to milliwatts.

Dialog Box
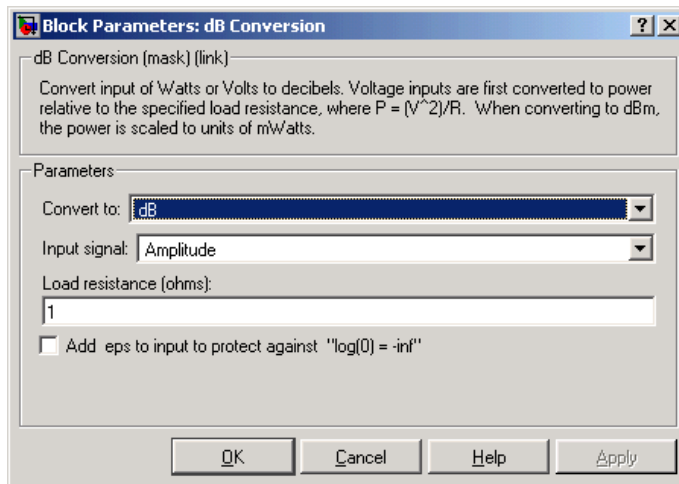
Convert to

   The logarithmic scaling to which the input is converted, dB or dBm. Tunable.

Input signal

   The type of input signal, Power or Amplitude. Nontunable.

Load resistance

   The scale factor used to convert voltage inputs to units of power. Tunable.

dB Conversion

Add eps to input to protect against "log(0) = -inf"
When selected, adds eps to all input values (power or voltage). Tunable.

Supported Data Types

- Double-precision floating point
- Single-precision floating point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

dB Gain	Signal Processing Blockset
Math Function	Simulink
log10	MATLAB

Purpose   Apply decibel gain

Library   Math Functions / Math Operations
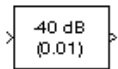
Description   The dB Gain block multiplies the input by the decibel values specified

in the **Gain** parameter. For an *M*-by-*N* input matrix u with elements u_{ij}, the **Gain** parameter can be a real *M*-by-*N* matrix with elements g_{ij} to be multiplied element-wise with the input, or a real scalar.

$$y_{ij} = 10u_{ij}^{(g_{ij}/k)}$$

The value of k is 10 for power signals (select Power as the **Input signal** parameter) and 20 for voltage signals (select Amplitude as the **Input signal** parameter).

The value of the equivalent linear gain

$$g_{ij}^{lin} = 10^{(g_{ij}/k)}$$

is displayed in the block icon below the dB gain value. The size and frame status of the output are the same as the input.

The dB Gain block supports real and complex floating-point and fixed-point data types.

Fixed-Point Data Types

The following diagram shows the data types used within the dB Gain subsystem block for fixed-point signals.
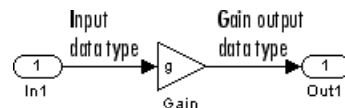
The settings for the fixed-point parameters of the Gain block in the diagram above are as follows:

• Round integer calculations toward: Floor

- Saturate on integer overflow — unselected

- Parameter data type mode — `Inherit via internal rule`

- Output data type mode — `Inherit via internal rule`

Refer to the Gain reference page for more information.

Dialog Box
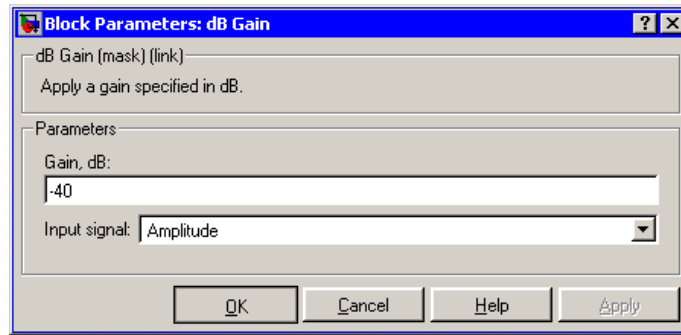
Gain

   The dB gain to apply to the input, a scalar or a real *M*-by-*N* matrix. Tunable.

Input signal

   The type of input signal: `Power` or `Amplitude`. Tunable.

Supported Data Types

- Double-precision floating point

- Single-precision floating point

- Fixed point (signed and unsigned)

- 8-, 16-, and 32-bit signed integers

- 8-, 16-, and 32-bit unsigned integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

dB Conversion	Signal Processing Blockset
Math Function	Simulink
log10	MATLAB

DCT

Purpose   Compute discrete cosine transform (DCT) of input

Library   Transforms

Description   The DCT block computes the unitary discrete cosine transform (DCT) of each channel in the *M*-by-*N* input matrix, *u*.

DCT

```
y = dct(u)     % Equivalent MATLAB code
```

For both sample-based and frame-based inputs, the block assumes that each input column is a frame containing *M* consecutive samples from an independent channel. The frame size, *M*, must be a power of two. To work with other frame sizes, use the Zero Pad block to pad or truncate the frame size to a power-of-two length.

The output is an *M*-by-*N* matrix whose *l*th column contains the length-*M* DCT of the corresponding input column.

$$y(k,l) = w(k) \sum_{m=1}^{M} u(m,l)\cos\frac{\pi(2m-1)(k-1)}{2M}, \qquad k = 1, ..., M$$

where

$$w(k) = \begin{cases} \dfrac{1}{\sqrt{M}}, & k = 1 \\ \sqrt{\dfrac{2}{M}}, & 2 \le k \le M \end{cases}$$

The output is always sample based, and the output port rate and data type (real/complex) are the same as those of the input port.

For convenience, length-*M* 1-D vector inputs and *sample-based* length-*M* row vector inputs are processed as single channels (that is, as *M*-by-1 column vectors), and the output has the same dimension as the input.

The **Sine and cosine computation** parameter determines how the block computes the necessary sine and cosine values. This parameter

has two settings, each with its advantages and disadvantages, as described in the following table.

Sine and Cosine Computation Parameter Setting	Sine and Cosine Computation Method	Effect on Block Performance
Table lookup	The block computes and stores the trigonometric values before the simulation starts, and retrieves them during the simulation. When you generate code from the block, the processor running the generated code stores the trigonometric values computed by the block in a speed-optimized table, and retrieves the values during code execution.	The block usually runs much more quickly, but requires extra memory for storing the precomputed trigonometric values.
Trigonometric fcn	The block computes sine and cosine values during the simulation. When you generate code from the block, the processor running the generated code computes the sine and cosine values while the code runs.	The block usually runs more slowly, but does not need extra data memory. For code generation, the block requires a support library to emulate the trigonometric functions, increasing the size of the generated code.

This block supports Simulink virtual buses.

Fixed-Point Data Types

The diagrams below show the data types used within the DCT block for fixed-point signals. You can set the sine table, accumulator, product output, and output data types displayed in the diagrams in the DCT block dialog as discussed in "Dialog Box" on page 10-203.

Inputs to the DCT block are first cast to the output data type and stored in the output buffer. Each butterfly stage processes signals in the accumulator data type, with the final output of the butterfly being cast back into the output data type.

Butterfly Stage Data Types

Twiddle Multiplication Data Types

The output of the multiplier is in the product output data type when at least one of the inputs to the multiplier is real. When both of the inputs to the multiplier are complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed, see "Multiplication Data Types" on page 8-16.

Dialog Box
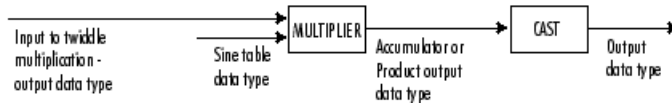
The **Main** pane of the DCT block dialog appears as follows:

Sine and cosine computation

Sets the block to compute sines and cosines by either looking up sine and cosine values in a speed-optimized table (Table lookup), or by making sine and cosine function calls (Trigonometric fcn). See the previous table.

The **Fixed-point** pane of the DCT block dialog appears as follows:

Rounding mode

Select the rounding mode for fixed-point operations. The sine table values do not obey this parameters; they always round to Nearest.

Overflow mode

Select the overflow mode for fixed-point operations. The sine table values do not obey this parameters; they always round to Nearest.

Sine table

Choose how you will specify the word length of the values of the sine table. The fraction length of the sine table values is always equal to the word length minus one:

- When you select `Same word length as input`, the word length of the sine table values will match that of the input to the block.

- When you select `Specify word length`, you are able to enter the word length of the sine table values, in bits.

The sine table values do not obey the **Rounding mode** and **Overflow mode** parameters; they are always saturated and rounded to `Nearest`.

Product output

Use this parameter to specify how you would like to designate the product output word and fraction lengths. Refer to "Fixed-Point Data Types" on page 10-201 and "Multiplication Data Types" on page 8-16 for illustrations depicting the use of the product output data type in this block:

- When you select `Inherit via internal rule`, the product output word length and fraction length are automatically set according to the following equations:

$$product\ output\ word\ length\ = \\ output\ word\ length + sine\ table\ values\ word\ length$$

$$product\ output\ fraction\ length\ = \\ output\ fraction\ length + sine\ table\ values\ fraction\ length$$

- When you select `Same as input`, these characteristics will match those of the input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the product output, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the product output. This block requires power-of-two slope and a bias of zero.

Accumulator

Use this parameter to specify how you would like to designate the accumulator word and fraction lengths. Refer to "Fixed-Point Data Types" on page 10-201 and "Multiplication Data Types" on page 8-16 for illustrations depicting the use of the accumulator data type in this block:

- When you select Inherit via internal rule, the accumulator word length and fraction length are automatically set according to the following equations:

$$accumulator\ word\ length\ =\ product\ output\ word\ length + 1$$

$$accumulator\ fraction\ length\ =\ product\ output\ fraction\ length$$

- When you select Same as product output, these characteristics will match those of the product output.

- When you select Same as input, these characteristics will match those of the input to the block.

- When you select Binary point scaling, you are able to enter the word length and the fraction length of the accumulator, in bits.

- When you select Slope and bias scaling, you are able to enter the word length, in bits, and the slope of the accumulator. This block requires power-of-two slope and a bias of zero.

Output

Choose how you will specify the output word length and fraction length:

- When you select Inherit via internal rule, the output word length and fraction length are automatically set according to the following equations:

$$output\ word\ length\ =\ input\ word\ length \\ +\ floor(\log 2(DCT\ length - 1)) + 1$$

$$output\ fraction\ length\ =\ input\ fraction\ length$$

- When you select Same as input, these characteristics will match those of the input to the block.

- When you select Binary point scaling, you are able to enter the word length and the fraction length of the output, in bits.

- When you select Slope and bias scaling, you are able to enter the word length, in bits, and the slope of the output. This block requires power-of-two slope and a bias of zero.

Supported Data Types

- Double-precision floating point

- Single-precision floating point

- Fixed point (signed only)

- 8-, 16-, and 32-bit signed integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Complex Cepstrum	Signal Processing Blockset
FFT	Signal Processing Blockset
IDCT	Signal Processing Blockset
Real Cepstrum	Signal Processing Blockset
dct	Signal Processing Toolbox

Delay

Purpose　　　Delay discrete-time input by specified number of samples or frames

Library　　　Signal Operations
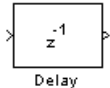
Description　The Delay block delays a discrete-time input by the number of samples or frames specified in the **Delay units** and **Delay** parameters. The **Delay** value must be an integer value greater than or equal to zero. Also, when you enter a value of zero for the **Delay** parameter, any initial conditions you might have entered have no effect on the output.

The Delay block allows you to set the initial conditions of the signal that is being delayed. The initial conditions must be numeric. Select the **Show additional parameters** check box in order to specify the initial conditions.

This block reference contains the following topics:

- "Sample-Based Operation" on page 10-208 — Use the Delay block with a sample-based input signal
- "Frame-Based Operation" on page 10-209 — Use the Delay block with a frame-based input signal

Sample-Based Operation

When the input is a sample-based M-by-N matrix, where $M \geq 1$ and $N \geq 1$, the block treats each of the $M*N$ matrix elements as an independent channel.

When the input is a sample-based scalar, the **Delay** parameter can be a scalar integer by which to equally delay all channels. When the input is a sample-based vector, the **Delay** parameter can be a scalar integer by which to equally delay all channels, or a vector whose length is equal to the number of channels. When the input is a sample-based M-by-N matrix, where $M>1$ and $N>1$, then the **Delay** parameter can be a scalar integer by which to equally delay all channels or an M-by-N matrix of nonnegative integers that specify the number of sample intervals to delay each channel of the input.

There are four different choices for initial conditions. The initial conditions can be the same or different for each channel. They can also be the same or different along each channel.

Frame-Based Operation

When the input is a frame-based *M*-by-*N* matrix, the block treats each of the *N* columns as an independent channel, and delays each channel as specified by the **Delay** parameter.

When the input is frame based, the **Delay** parameter can be a scalar integer by which to equally delay all channels or a vector whose length is equal to the number of channels.

There are four different choices for initial conditions. The initial conditions can be the same or different for each channel. They can also be constant or varying along each channel.

Resetting the Delay

The Delay block resets the delay whenever it detects a reset event at the optional Rst port. The reset signal rate must be a positive integer multiple of the rate of the data signal input.

The reset event is specified by the **Reset port** parameter, and can be one of the following:

- None disables the Rst port.

- Rising edge triggers a reset operation when the Rst input does one of the following:

  - Rises from a negative value to a positive value or zero

  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)
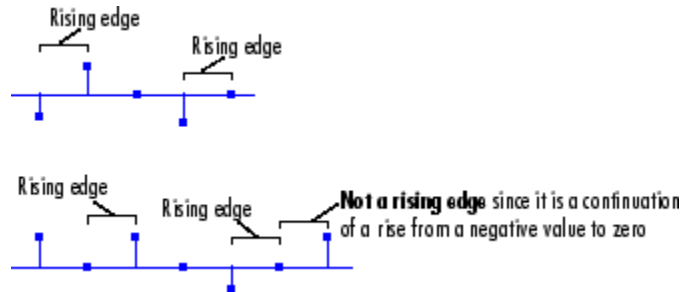
- `Falling edge` triggers a reset operation when the Rst input does one of the following:

  - Falls from a positive value to a negative value or zero

  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero (see the following figure)
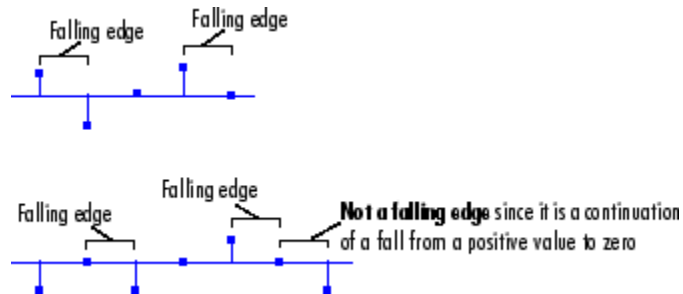
- `Either edge` triggers a reset operation when the Rst input is Rising edge or Falling edge (as described above).

- `Non-zero sample` triggers a reset operation at each sample time that the Rst input is not zero.

Note When running simulations in the Simulink `MultiTasking` mode, reset signals have a one-sample latency. Therefore, when the block detects a reset event, there is a one-sample delay at the reset port rate before the block applies the reset. For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" on page 2-56 and "Models with Multiple Sample Rates" in the Real-Time Workshop User's Guide documentation.

This block supports Simulink virtual buses.

Dialog Box
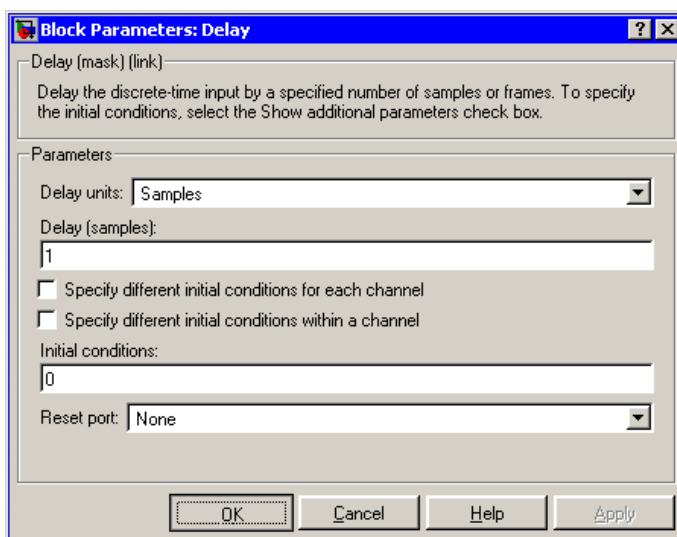
Delay units

Select whether you want to delay your input by a specified number of `Samples` or `Frames`. You can choose to delay your signal by a certain number of samples or frames regardless of whether your input is sample or frame based.

Delay

Delay (samples) or Delay (frames)
See "Sample-Based Operation" on page 10-208 and "Frame-Based Operation" on page 10-209 for a description of what format to use for each configuration of the block dialog.

Specify different initial conditions for each channel
Select this check box when you want the initial conditions to vary across the channels. When you do not select this check box, the initial conditions are the same across the channels.

Specify different initial conditions within a channel
Select this check box when you want the initial conditions to vary within the channels. When you do not select this check box, the initial conditions are the same within the channels.

Initial conditions
Enter a scalar, vector, matrix, or cell array of initial condition values depending on your choice for the **Specify different initial conditions for each channel** and **Specify different initial conditions within a channel** check boxes. See "Sample-Based Operation" on page 10-208 and "Frame-Based Operation" on page 10-209 for a description of what format to use for each configuration of the block dialog.

Reset port
Determines the reset event that causes the block to reset the delay. For more information, see "Resetting the Delay" on page 10-209.

Examples

Sample-Based Operation Examples

There are four different choices for initial conditions. The initial conditions can be the same or different for each channel. They can also be the same or different along each channel. The next sections describe the behavior of the block for each of these four cases:

- "Case 1 — Use the Same Initial Conditions for Each Channel and Within a Channel" on page 10-213

- "Case 2 — Use Different Initial Conditions for Each Channel and the Same Initial Conditions Within a Channel" on page 10-214

Case 1 — Use the Same Initial Conditions for Each Channel and Within a Channel

Enter a scalar value for the initial conditions. This value is used as the constant initial condition value for each of the channels.

For example, suppose your input is a sample-based matrix.

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}, \begin{bmatrix} 3 & 3 \\ 3 & 3 \end{bmatrix}, \ldots$$

You want the initial conditions of your four-channel signal to be identical and zero for the first two samples:

1 For the **Delay (samples)** parameter, type 2.

2 Clear the **Specify different initial conditions for each channel** and **Specify different initial conditions within a channel** check boxes.

3 For the **Initial conditions** parameter, specify a scalar value of 0.

The output of the delay block is

$$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}, \begin{bmatrix} 3 & 3 \\ 3 & 3 \end{bmatrix}, \ldots$$

Note how 0, the scalar initial condition value, is used for each channel and within the channels. It is the output at sample time zero and sample time one.

Case 2 — Use Different Initial Conditions for Each Channel and the Same Initial Conditions Within a Channel

The initial conditions can be either a matrix for matrix input or a vector for vector input. These initial condition values are used as the constant initial condition value for each of the channels.

For example, suppose your input is a sample-based matrix.

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}, \begin{bmatrix} 3 & 3 \\ 3 & 3 \end{bmatrix}, \cdots$$

You want the initial conditions of your four-channel signal to be

$$\begin{bmatrix} 7 & 9 \\ 11 & 13 \end{bmatrix}$$

for the first two samples:

1 For the **Delay (samples)** parameter, type 2.

2 Select the **Specify different initial conditions for each channel** check box.

3 Clear the **Specify different initial conditions within a channel** check box.

4 For the **Initial conditions** parameter, type [7 9; 11 13].

The output of the delay block is

$$\begin{bmatrix} 7 & 9 \\ 11 & 13 \end{bmatrix}, \begin{bmatrix} 7 & 9 \\ 11 & 13 \end{bmatrix}, \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}, \begin{bmatrix} 3 & 3 \\ 3 & 3 \end{bmatrix}, \cdots$$

Note how the initial condition matrix is the output at sample time zero and sample time one. Different initial conditions are used for each channel; the same initial condition value is used within a channel.

Case 3 — Use the Same Initial Conditions for Each Channel and Different Initial Conditions Within a Channel

In this case, when the input is a sample-based vector, the **Delay** parameter can be a scalar integer by which to equally delay all channels or a vector whose length is equal to the number of channels. All the values of this vector must be equal.

Enter the initial conditions as a vector, where the vector length is equal to the delay value. These values are used as the initial condition value along each of the channels to be delayed.

For example, suppose your input is a sample-based matrix.

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}, \begin{bmatrix} 3 & 3 \\ 3 & 3 \end{bmatrix}, \dots$$

You want the initial conditions of your four channel signal to be the same along each of the channels to be delayed:

1 For the **Delay (samples)** parameter, type 2.

2 Clear the **Specify different initial conditions for each channel** check box.

3 Select the **Specify different initial conditions within a channel** check box.

4 For the **Initial conditions** parameter, type [10 20].

The output of the delay block is

$$\begin{bmatrix} 10 & 10 \\ 10 & 10 \end{bmatrix}, \begin{bmatrix} 20 & 20 \\ 20 & 20 \end{bmatrix}, \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}, \begin{bmatrix} 3 & 3 \\ 3 & 3 \end{bmatrix}, \dots$$

Note how the first element of the initial conditions vector is the output, for all channels, at sample time zero. The second element of the initial conditions vector is the output, for all channels, at sample

time one. The same initial conditions are used for each channel, but different initial condition values are used with a channel.

Case 4 — Use Different Initial Conditions for Each Channel and Within a Channel

Enter a cell array for your initial condition values. Each cell of the cell array represents the delay values for one channel. The cell array must have the same size as your input signal. Or, when you have a nonmatrix input and a scalar delay value, you can enter the initial conditions as a matrix.

For example, suppose your input is a sample-based vector.

$$\begin{bmatrix} 1 & 1 \end{bmatrix}, \begin{bmatrix} 2 & 2 \end{bmatrix}, \begin{bmatrix} 3 & 3 \end{bmatrix}, \cdots$$

You want the initial conditions of your two channel signal to be different for each channel and along each channel:

1 For the **Delay (samples)** parameter, type 2.

2 Select the **Specify different initial conditions for each channel** and **Specify different initial conditions within a channel** check boxes.

3 For the **Initial conditions** parameter, type [10 20; 30 40]

The output of the delay block is

$$\begin{bmatrix} 10 & 20 \end{bmatrix}, \begin{bmatrix} 30 & 40 \end{bmatrix}, \begin{bmatrix} 1 & 1 \end{bmatrix}, \begin{bmatrix} 2 & 2 \end{bmatrix} \cdots$$

Note that the first row of the initial conditions vector is the output at sample time zero. The second row of the initial conditions vector is the output at sample time one. Different initial conditions are used for each channel and within the channels.

In addition, suppose your input is a sample-based matrix.

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}, \begin{bmatrix} 3 & 3 \\ 3 & 3 \end{bmatrix}, \cdots$$

You want the initial conditions of your two-channel signal to be different for each channel and along each channel.

1 For the **Delay (samples)** parameter, type 2.

2 Select the **Specify different initial conditions for each channel** and the **Specify different initial conditions within a channel** check boxes.

3 For the **Initial conditions** parameter, type {[11 15] [12 16]; [13 17] [14 18]}. Note that the dimensions of the cell array match the dimensions of the input. Also, each element of the cell array represents the initial conditions within one channel.

The output of the delay block is

$$\begin{bmatrix} 11 & 12 \\ 13 & 14 \end{bmatrix}, \begin{bmatrix} 15 & 16 \\ 17 & 18 \end{bmatrix}, \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}, \cdots$$

Note how each element of the cell array represents the initial conditions within a channel. The first element, a vector, represents the initial conditions within channel 1. The second element, a vector, represents the initial conditions within channel 2, and so on. Different initial conditions are used for each channel and within the channels.

Frame-Based Operation Examples

There are four different choices for initial conditions. The initial conditions can be the same or different for each channel. They can also be constant or varying along each channel. The next sections describe the behavior of the block for each of these four cases:

- "Case 1 — Use the Same Initial Conditions for Each Channel and Within a Channel" on page 10-218

Delay

Case 1 — Use the Same Initial Conditions for Each Channel and Within a Channel

Enter a scalar value for the initial conditions. This value is used as the constant initial condition value for each of the channels.

For example, suppose your input is a frame-based matrix.

$$\begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 & 4 \\ 5 & 5 & 5 \\ 6 & 6 & 6 \end{bmatrix}, \begin{bmatrix} 7 & 7 & 7 \\ 8 & 8 & 8 \\ 9 & 9 & 9 \end{bmatrix}, \cdots$$

You want the initial conditions of your three-channel signal to be identical and zero for the first frame:

1 For the **Delay (frames)** parameter, type 1.

2 Clear the **Specify different initial conditions for each channel** and the **Specify different initial conditions within a channel** check boxes.

3 For the **Initial conditions** parameter, specify a scalar value of 0.

The output of the delay block is

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 & 4 \\ 5 & 5 & 5 \\ 6 & 6 & 6 \end{bmatrix}, \begin{bmatrix} 7 & 7 & 7 \\ 8 & 8 & 8 \\ 9 & 9 & 9 \end{bmatrix}, \cdots$$

Note how 0, the scalar initial condition value, is used across the channels and within the channels for the first frame. This frame is the output at sample time zero.

Case 2 — Use Different Initial Conditions for Each Channel and the Same Initial Conditions Within a Channel

The initial conditions must be a vector of length N, where $N \geq 1$. N is also equal to the number of channels in your signal. These initial condition values are used as the constant initial condition value for each of the channels.

For example, suppose your input is a frame-based matrix.

$$\begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 & 4 \\ 5 & 5 & 5 \\ 6 & 6 & 6 \end{bmatrix}, \begin{bmatrix} 7 & 7 & 7 \\ 8 & 8 & 8 \\ 9 & 9 & 9 \end{bmatrix}, \cdots$$

You want the initial conditions of your three-channel signal to be [0 10 20] for the first frame:

1 For the **Delay (frames)** parameter, type 1.

2 Select the **Specify different initial conditions for each channel** check box.

3 Clear the **Specify different initial conditions within a channel** check box.

4 For the **Initial conditions** parameter, type [0 10 20].

The output of the delay block is

$$\begin{bmatrix} 0 & 10 & 20 \\ 0 & 10 & 20 \\ 0 & 10 & 20 \end{bmatrix}, \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 & 4 \\ 5 & 5 & 5 \\ 6 & 6 & 6 \end{bmatrix}, \begin{bmatrix} 7 & 7 & 7 \\ 8 & 8 & 8 \\ 9 & 9 & 9 \end{bmatrix}, \cdots$$

Note how the initial condition vector is expanded to create the frame that is output at sample time zero. Different initial conditions are used for each channel, but the same initial condition value is used with a channel.

Case 3 — Use the Same Initial Conditions for Each Channel and Different Initial Conditions Within a Channel

In this case, the **Delay** parameter can be a scalar integer by which to equally delay all channels or a vector whose length is equal to the number of channels. All the values of this vector must be equal.

Enter the initial conditions as a vector. These values are used as the initial condition value along each of the channels to be delayed. The initial condition vector must have length equal to the value of the **Delay (frames)** parameter multiplied by the frame length. For example, if you want to delay your signal by two frames with frame length two and an initial condition value of 3, enter your initial condition vector as [3 3 3 3].

For example, suppose your input is a frame-based matrix.

$$\begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 & 4 \\ 5 & 5 & 5 \\ 6 & 6 & 6 \end{bmatrix}, \begin{bmatrix} 7 & 7 & 7 \\ 8 & 8 & 8 \\ 9 & 9 & 9 \end{bmatrix}, \cdots$$

You want the initial conditions of your three-channel signal to be the same along each of the channels to be delayed:

1 For the **Delay (frame)** parameter, type 1.

2 Clear the **Specify different initial conditions for each channel** check box.

3 Select the **Specify different initial conditions within a channel** check box.

4 For the **Initial conditions** parameter, type [10 20 30].

The output of the delay block is

$$\begin{bmatrix} 10 & 10 & 10 \\ 20 & 20 & 20 \\ 30 & 30 & 30 \end{bmatrix}, \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 & 4 \\ 5 & 5 & 5 \\ 6 & 6 & 6 \end{bmatrix}, \begin{bmatrix} 7 & 7 & 7 \\ 8 & 8 & 8 \\ 9 & 9 & 9 \end{bmatrix}, \cdots$$

Note how the initial condition vector defines the initial condition values within each of the three channels. The same initial conditions are used for each channel, but different initial condition values are used with a channel.

Case 4 — Use Different Initial Conditions for Each Channel and Within a Channel

Enter a cell array for your initial condition values. Or, when you have a scalar delay value, you can enter the initial conditions as a matrix.

For example, suppose your input is a frame-based matrix.

$$\begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 & 4 \\ 5 & 5 & 5 \\ 6 & 6 & 6 \end{bmatrix}, \begin{bmatrix} 7 & 7 & 7 \\ 8 & 8 & 8 \\ 9 & 9 & 9 \end{bmatrix}, \cdots$$

You want the initial conditions of your three-channel signal to be different for each channel and along each channel.

1 For the **Delay (frames)** parameter, type 1.

2 Select the **Specify different initial conditions for each channel** and the **Specify different initial conditions within a channel** check boxes.

3 For the **Initial conditions** parameter, type either [10 20 30; 40 50 60; 70 80 90] or {[10 40 70];[20 50 80];[30 60 90]}. Note that each cell of the cell array represents the delay along one channel.

Regardless of whether you use a matrix or cell array, the output of the delay block is

Delay

$$\begin{bmatrix} 10 & 20 & 30 \\ 40 & 50 & 60 \\ 70 & 80 & 90 \end{bmatrix}, \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 & 4 \\ 5 & 5 & 5 \\ 6 & 6 & 6 \end{bmatrix}, \begin{bmatrix} 7 & 7 & 7 \\ 8 & 8 & 8 \\ 9 & 9 & 9 \end{bmatrix} \cdots$$

Note how the initial condition matrix is the output at sample time zero. The elements of the initial condition cell array define the initial condition values within each channel. The first element, a vector, represents the initial conditions within channel 1. The second element, a vector, represents the initial conditions within channel 2, and so on. Different initial conditions are used for each channel and within the channels.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point (signed and unsigned)
- Custom data types
- Boolean
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Unit Delay	Simulink
Variable Fractional Delay	Signal Processing Blockset
Variable Integer Delay	Signal Processing Blockset

Purpose   Rebuffer sequence of inputs with one-sample shift

Library   Signal Management / Buffers

Description   The Delay Line block buffers the input samples into a sequence of overlapping or underlapping matrix outputs. In the most typical use (sample-based inputs), each output differs from the preceding output by only one sample, as illustrated below for scalar input.
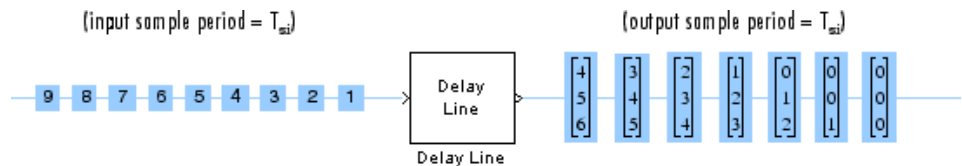
Note that the first output of the block in the example above is all zeros; this is because the **Initial Conditions** parameter is set to zero. Due to the latency of the Delay Line block, all outputs are delayed by one frame, the entries of which are defined by the **Initial Conditions** parameter.

Sample-Based Operation

In sample-based operation, the Delay Line block buffers a sequence of sample-based length-N vector inputs (1-D, row, or column) into a sequence of overlapping frame-based M_o-by-N matrix outputs, where M_o is specified by the **Delay line size** parameter ($M_o > 1$). That is, each input vector becomes a *row* in the frame-based output matrix.

At each sample time the new input vector is added in the last row of the output, so each output overlaps the previous output by M_o-1 samples. Therefore, the output sample period and frame period is the same as the input sample period ($T_{so} = T_{si}$, and $T_{fo} = T_{si}$). When $M_o = 1$, the input is simply passed through to the output and retains the same dimension, but becomes frame based. The latency of the block always causes an initial delay in the output; the value of the first output is specified by the **Initial conditions** parameter (see "Initial Conditions" on page 10-226). Sample-based full-dimension matrix inputs are not accepted.

Delay Line

The Delay Line block's sample-based operation is similar to that of a Buffer block with **Buffer size** equal to M_o and **Buffer overlap** equal to M_o-1, except that the Buffer block has a different latency.

In the following model, the block operates on a sample-based input with a **Delay line size** of 3.
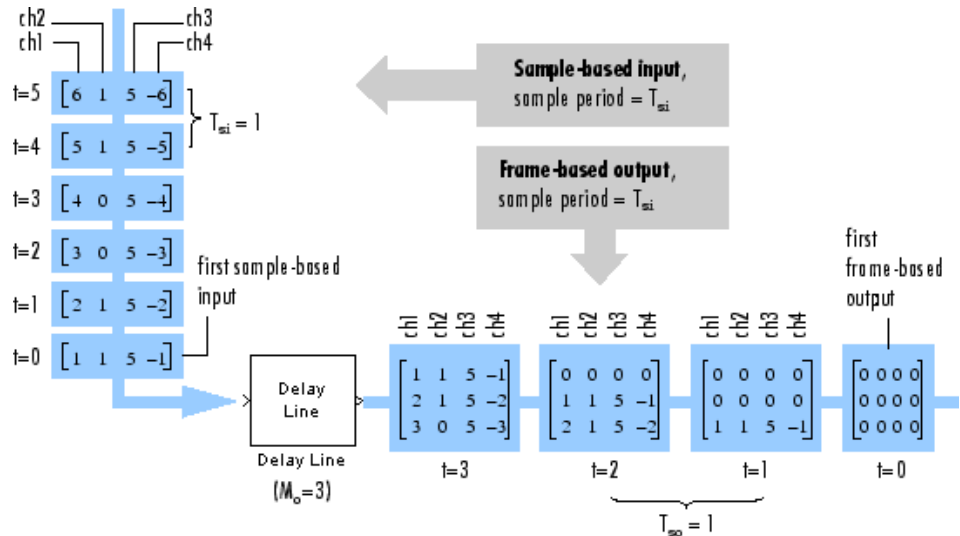
The input vectors in the example above do not begin appearing at the output until the second row of the second matrix due to the block's latency (see "Initial Conditions" on page 10-226). The first output matrix (all zeros in this example) reflects the block's **Initial conditions** setting. As for any sample-based input, the output frame rate and output sample rate are both equal to the input sample rate.

Frame-Based Operation

In frame-based operation, the Delay Line block rebuffers a sequence of frame-based M_i-by-N matrix inputs into a sequence of frame-based M_o-by-N matrix outputs, where M_o is the output frame size specified by the **Delay line size** parameter. Depending on whether M_o is greater than, less than, or equal to the input frame size, M_i, the output frames

can be underlapped or overlapped. Each of the N input channels is rebuffered independently.

When $M_o > M_i$, the output frame overlap is the difference between the output and input frame size, M_o-M_i. When $M_o < M_i$, the output is underlapped; the Delay Line block discards the first M_i-M_o samples of each input frame so that only the last M_o samples are buffered into the corresponding output frame. When $M_o = M_i$, the output data is identical to the input data, but is delayed by the latency of the block. Due to the block's latency, the outputs are always delayed by one frame, the entries of which are specified by the **Initial conditions** (see "Initial Conditions" on page 10-226).

The output frame period is equal to the input frame period (T_{fo}=T_{fi}). The output sample period, T_{so}, is therefore equal to T_{fi}/M_o, or equivalently, $T_{si}(M_i/M_o)$

In the following model, the block rebuffers a two-channel frame-based input with a **Delay line size** of 3.
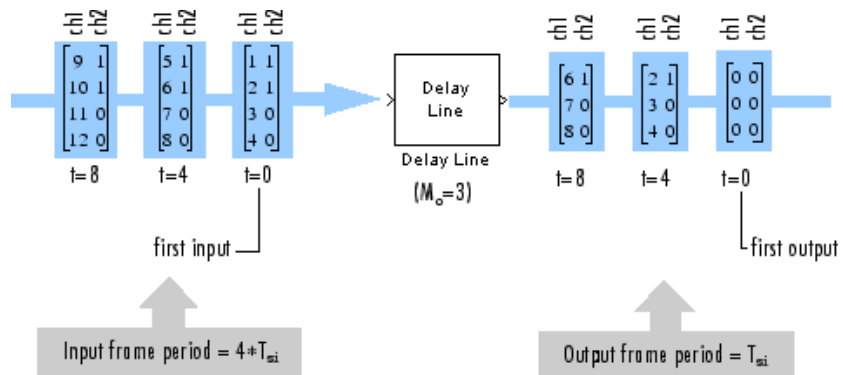
The first output frame in the example is a product of the latency of the Delay Line block; it is all zeros because the **Initial conditions** is set to be zero. Since the input frame size, 4, is larger than the output frame size, 3, only the last three samples in each input frame are propagated to the corresponding output frame. The frame periods of the input and

output are the same, and the output sample period is $T_{si}(M_i/M_o)$, or 4/3 the input sample period.

Initial Conditions

The Delay Line block's buffer is initialized to the value specified by the **Initial condition** parameter. The block outputs this buffer at the first simulation step (*t*=0). When the block's output is a vector, the **Initial condition** can be a vector of the same size, or a scalar value to be repeated across all elements of the initial output. When the block's output is a matrix, the **Initial condition** can be a matrix of the same size, a vector (of length equal to the number of matrix rows) to be repeated across all columns of the initial output, or a scalar to be repeated across all elements of the initial output.

Dialog Box
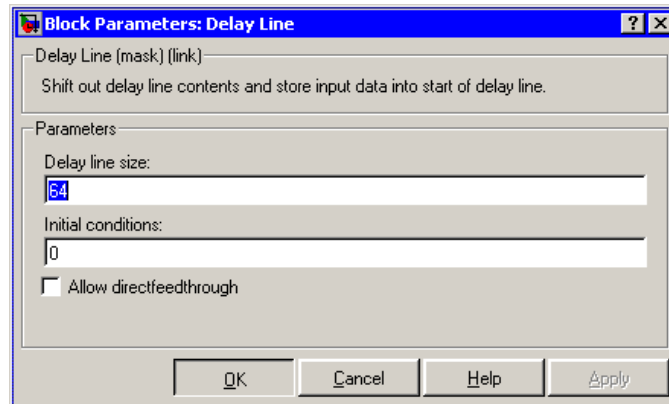
Delay line size
> The number of rows in output matrix, M_o.

Initial conditions
> The value of the block's initial output, a scalar, vector, or matrix.

Allow direct feedthrough
> When you select this check box, the input data is not delayed by an extra frame before it is available at the output buffer. Instead, the input data is available immediately at the output port of the block.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Custom data types • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Custom data types • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Buffer	Signal Processing Blockset
Triggered Delay Line	Signal Processing Blockset

Detrend

Purpose   Remove linear trend from vectors

Library   Statistics
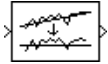
Description   The Detrend block removes a linear trend from the length-*M* input
                     vector, *u*, by subtracting the straight line that best fits the data in the
                     least squares sense.

                     The least squares line, $\hat{u} = ax + b$, is the line with parameters *a* and *b*
                     that minimizes the quantity

                     $$\sum_{i=1}^{M} (u_i - \hat{u}_i)^2$$

                     for *M* evenly-spaced values of *x*, where u_i is the *i*th element in the input
                     vector. The output, $y = u\text{-}\hat{u}$, is an *M*-by-1 column vector (regardless of
                     the input vector dimension) with the same frame status as the input.
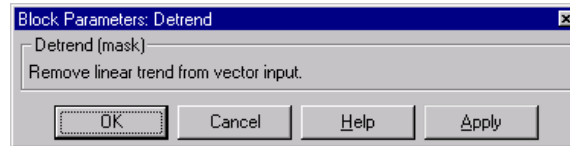
**Dialog
Box**

**Supported
Data
Types**

• Double-precision floating point

• Single-precision floating point

To learn how to convert your data types to the above data types in
MATLAB and Simulink, see "Supported Data Types and How to Convert
to Them" on page 7-2.

See Also

Cumulative Sum	Signal Processing Blockset
Difference	Signal Processing Blockset

Least Squares Polynomial Fit	Signal Processing Blockset
Unwrap	Signal Processing Blockset
detrend	MATLAB

Difference

Purpose   Compute element-to-element difference along rows or columns

Library   Math Functions / Math Operations

Description   The Difference block computes the difference between adjacent elements in rows or columns of the *M*-by-*N* input matrix u.
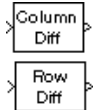
Columnwise Differencing

When the **Difference along** parameter is set to `Columns`, the block computes differences between adjacent column elements.

```
y = diff(u)      % Equivalent MATLAB code
```

For sample-based inputs, the output is a sample-based (*M*-1)-by-*N* matrix whose *j*th column has elements

$$y_{i,j} = u_{i+1,j} - u_{i,j} \qquad 1 \leq i \leq (M-1)$$

For convenience, length-*M* 1-D vector inputs are treated as *M*-by-1 column vectors for columnwise differencing, and the output is 1-D.

For example, the following figure shows the block output for sample-based inputs:
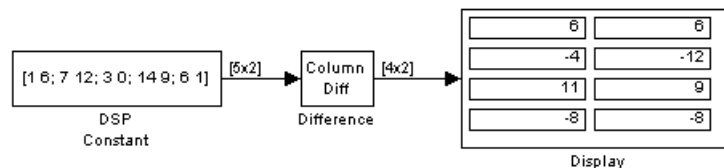
For frame-based inputs, the output is a frame-based *M*-by-*N* matrix whose *j*th column has elements

$$y_{i,j} = u_{i+1,j} - u_{i,j} \qquad 2 \leq i \leq (M-1)$$

The first row of the first output contains the difference between the first row of the first input and zero. The first row of each subsequent output

contains the difference between the first row of the current input (time
t) and the last row of the previous input (time t-T_{f}).

$$'_{1,j}(t) = u_{1,j}(t) - u_{M,j}(t - T_f)$$

For example, the following figure shows the block output for
frame-based inputs:
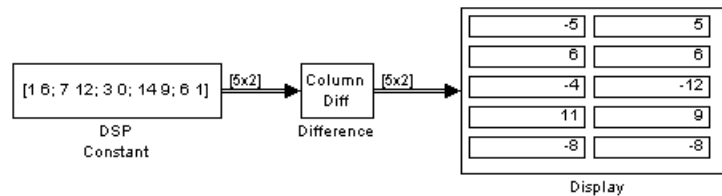
Rowwise Differencing

When the **Difference along** parameter is set to Rows, the block
computes differences between adjacent row elements. The result is the
same regardless of the frame status of the input signal.

```
y = diff(u,[],2)        % Equivalent MATLAB code
```

The output is an *M*-by-(*N*-1) matrix whose *i*th row has elements

$$'_{i,j} = u_{i,j+1} - u_{i,j} \qquad 1 \le j \le (N-1)$$

The frame status of the output is the same as the input. For
convenience, length-*N* 1-D vector inputs are treated as 1-by-*N* row
vectors for rowwise differencing, and the output is 1-D.

For example, the following figure shows the block output for
sample-based inputs. The output is the same for frame-based inputs:

Difference
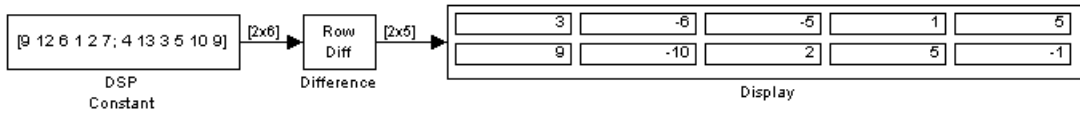
Fixed-Point Data Types

The following diagram shows the data types used within the Difference block for fixed-point signals.
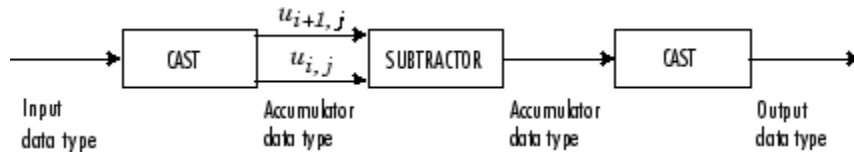
You can set the accumulator and output data types in the block dialog as discussed in "Dialog Box" on page 10-233 .

Dialog Box

The **Main** pane of the Difference block appears as follows:
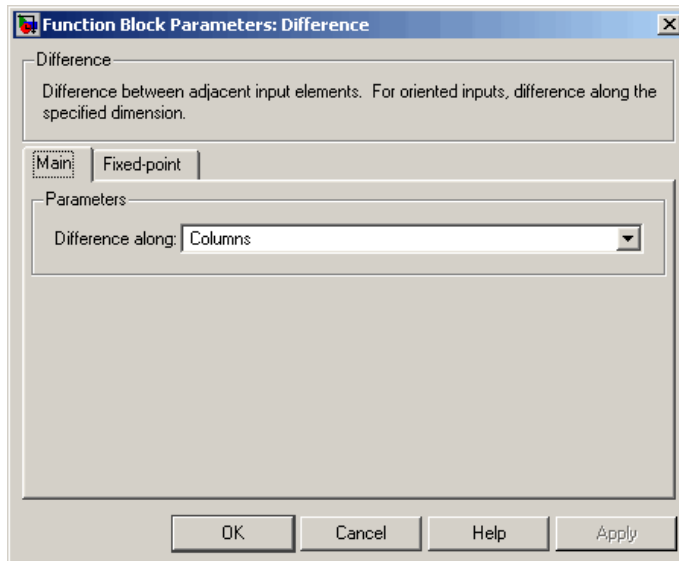
Difference along

Specify the dimension along which to compute element-to-element differences. Columns specifies columnwise differencing, while Rows specifies rowwise differencing. Nontunable.

Difference

The **Fixed-point** pane of the Difference block appears as follows:

Rounding mode

Select the rounding mode for fixed-point operations.

Overflow mode

Select the overflow mode for fixed-point operations.
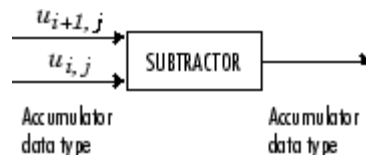
Accumulator

Use this parameter to specify how you would like to designate the accumulator word and fraction lengths:

- When you select `Inherit via internal rule`, the accumulator word length and fraction length are automatically set according to the following equations:

$$ideal\ accumulator\ word\ length\ =\ input\ word\ length\ +\ 1$$

$$ideal\ accumulator\ fraction\ length\ =\ input\ fraction\ length$$

> **Note** The actual accumulator word length may be equal to or greater than the calculated ideal product output word length, depending on the settings on the **Hardware Implementation** pane of the Configuration Parameter dialog box.

- When you select `Same as input`, these characteristics will match those of the input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and fraction length of the accumulator, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the accumulator. This block requires power-of-two slope and a bias of zero.

Output

Choose how you will specify the output word length and fraction length:

- When you select `Same as accumulator`, these characteristics will match those of the accumulator.

- When you select `Same as input`, these characteristics will match those of the input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and fraction length of the output, in bits.

Difference

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the output. This block requires power-of-two slope and a bias of zero.

Lock scaling against changes by the autoscaling tool
Select this parameter to prevent any fixed-point scaling you specify in this block mask from being overridden by the autoscaling tool in the Fixed-Point Settings interface. For more information about the autoscaling tool, refer to "Fixed-Point Settings Interface" on page 8-28.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point (signed only)
- 8-, 16-, and 32-bit signed integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

| Cumulative Sum | Signal Processing Blockset |
| diff | MATLAB |

Purpose   Filter each channel of input over time using static or time-varying digital filter implementations

Library   Filtering / Filter Designs

Description

Note Use this block to efficiently implement a floating-point or fixed-point filter for which you know the coefficients, or that is already defined in a Signal Processing Toolbox `dfilt` object. The following Signal Processing Blockset blocks also implement digital filters, but serve slightly different purposes:

- Digital Filter Design — Use to design, analyze, and then efficiently implement floating-point filters. This block provides the same filter implementation as the Digital Filter block for floating-point signals.

- Filter Realization Wizard — Use to implement floating-point or fixed-point filters built from Sum, Gain, and Unit Delay blocks. You can either design the filter using block filter design and analysis parameters, or import the coefficients of a filter that you designed elsewhere.

The Digital Filter block independently filters each channel of the input signal with a specified digital IIR or FIR filter. The block can implement *static filters* with fixed coefficients, as well as *time-varying filters* with coefficients that change over time. You can tune the coefficients of a static filter during simulation.

This block filters each channel of the input signal independently over time. The output frame status and dimensions are always the same as those of the input signal that is filtered. When inputs are frame based, the block treats each column as an independent channel; the block filters each column. When inputs are sample based, the block treats each element of the input as an individual channel.

Digital Filter

The outputs of this block numerically match the outputs of the Digital Filter Design block and of the `dfilt` function in the Signal Processing Toolbox.

Sections of This Reference Page

- "Filter Source" on page 10-238
- "Supported Filter Structures" on page 10-238
- "Specifying Initial Conditions" on page 10-241
- "State Logging" on page 10-243
- "Fixed-Point Data Types" on page 10-244
- "Dialog Box" on page 10-245
- "Filter Structure Diagrams" on page 10-260
- "Supported Data Types" on page 10-295
- "See Also" on page 10-295

Filter Source

The Digital Filter block can operate in two different modes. Select the mode in the **Filter source** group box. If you select **Specify filter characteristics in dialog**, you enter information about the filter such as structure and coefficients in the block mask. If you select **Specify discrete-time filter object (DFILT)**, you specify the filter using a `dfilt` object from the Signal Processing Toolbox.

Supported Filter Structures

When you select **Specify discrete-time filter object (DFILT)**, the following `dfilt` structures are supported:

- `dfilt.df1`
- `dfilt.df1t`
- `dfilt.df2`

- dfilt.df2t

- dfilt.df1sos

- dfilt.df1tsos

- dfilt.df2sos

- dfilt.df2tsos

- dfilt.dffir

- dfilt.dffirt

- dfilt.dfsymfir

- dfilt.dfasymfir

- dfilt.latticear

- dfilt.latticemamin

When you select **Specify filter characteristics in dialog**, the list of filter structures offered in the **Filter structure** parameter depends on whether you set the **Transfer function type** to IIR (poles & zeros), IIR (all poles), or FIR (all zeros), as summarized in the following table.

Note Each structure listed in the table below supports both fixed-point and floating-point signals.

The table also shows the vector or matrix of filter coefficients you must provide for each filter structure. For more information on how to specify filter coefficients for various filter structures, see "Specifying Static Filters" on page 3-10 and "Specifying Time-Varying Filters" on page 3-11.

Digital Filter

Filter Structures and Filter Coefficients

Transfer Function Type	Supported Filter Structures	Filter Coefficient Specification
IIR (poles & zeros)	Direct form I Direct form I transposed Direct form II Direct form II transposed	• Numerator coefficients vector [b0, b1, b2, ..., bn] • Denominator coefficients vector [a0, a1, a2, ..., am]
	Biquadratic direct form I (SOS) Biquadratic direct form I transposed (SOS) Biquadratic direct form II (SOS) Biquadratic direct form II transposed (SOS)	• *M*-by-6 second-order section (SOS) matrix. • Scale values See "Specifying the SOS Matrix (Biquadratic Filter Coefficients)" on page 3-16.
IIR (all poles)	Direct form Direct form transposed	Denominator coefficients vector [a0, a1, a2, ..., am]
	Lattice AR	Reflection coefficients vector [k1, k2, ..., kn]
FIR (all zeros)	Direct form Direct form symmetric Direct form antisymmetric Direct form transposed	Numerator coefficients vector [b0, b1, b2, ..., bn]
	Lattice MA	Reflection coefficients vector [k1, k2, ..., kn]

Specifying Initial Conditions

In **Specify filter characteristics in dialog** mode, the block initializes the internal filter states to zero by default, which is equivalent to assuming past inputs and outputs are zero. You can optionally use the **Initial conditions** parameter to specify nonzero initial conditions for the filter delays.

To determine the number of initial condition values you must specify, and how to specify them, refer to the following table on Valid Initial Conditions and Number of Delay Elements (Filter States) on page 10-243. The **Initial conditions** parameter can take one of four forms as described in the following table.

Valid Initial Conditions

Initial Condition	Examples	Description
Scalar	5 Each delay element for each channel is set to 5.	The block initializes all delay elements in the filter to the scalar value.
Vector (for applying the same delay elements to each channel)	For a filter with two delay elements: $[d_1\ d_2]$ The delay elements for all channels are d1 and d2.	Each vector element specifies a unique initial condition for a corresponding delay element. The block applies the same vector of initial conditions to each channel of the input signal. The vector length must equal the number of delay elements in the filter (specified in the Number of Delay Elements (Filter States) table).

Digital Filter

Initial Condition	Examples	Description
Vector or matrix (for applying different delay elements to each channel)	For a 3-channel input signal and a filter with two delay elements: $[d_1\ d_2\ D_1\ D_2\ d_1\ d_2]$ or $$\begin{bmatrix} d_1 & D_1 & d_1 \\ d_2 & D_2 & d_2 \end{bmatrix}$$ • The delay elements for channel 1 are d1 and d2. • The delay elements for channel 2 are D_1 and D_2. • The delay elements for channel 3 are d_1 and d_2.	Each vector or matrix element specifies a unique initial condition for a corresponding delay element in a corresponding channel: • The vector length must be equal to the product of the number of input channels and the number of delay elements in the filter (specified in the Number of Delay Elements (Filter States) on page 10-243 table). • The matrix must have the same number of rows as the number of delay elements in the filter (specified in the Number of Delay Elements (Filter States) on page 10-243 table), and must have one column for each channel of the input signal.
Empty matrix	[ ] Each delay element for each channel is set to 0.	The empty matrix, [ ], is equivalent to setting the **Initial conditions** parameter to the scalar value 0.

The number of delay elements (filter states) per input channel depends on the filter structure, as indicated in the following table.

Number of Delay Elements (Filter States)

Filter Structure	Number of Delay Elements per Channel
Direct form Direct form transposed Direct form symmetric Direct form antisymmetric	#_of_filter_coeffs-1
Direct form I Direct form I transposed	• #_of_zeros-1 • #_of_poles-1
Direct form II Direct form II transposed	max(#_of_zeros, #_of_poles)-1
Biquadratic direct form I (SOS) Biquadratic direct form I transposed (SOS) Biquadratic direct form II (SOS) Biquadratic direct form II transposed (SOS)	2 * #_of_filter_sections
Lattice AR Lattice MA	#_of_reflection_coeffs

State Logging

Simulink enables you to log the states in your model to the MATLAB workspace. The following table indicates which filter structures of the Digital Filter block support the Simulink state logging feature. Refer to "States" in the Simulink User's Guide documentation for more information.

Digital Filter

Transfer Function Type	Filter Structure	State Logging Supported
IIR (poles & zeros)	Direct form I	No
	Direct form I transposed	Yes
	Direct form II	No
	Direct form II transposed	Yes
	Biquadratic direct form I (SOS)	Yes
	Biquadratic direct form I transposed (SOS)	Yes
	Biquadratic direct form II (SOS)	Yes
	Biquadratic direct form II transposed (SOS)	Yes
IIR (all poles)	Direct form	No
	Direct form transposed	Yes
	Lattice AR	Yes
FIR (all zeros)	Direct form	No
	Direct form symmetric	No
	Direct form antisymmetric	No
	Direct form transposed	Yes
	Lattice MA	Yes

Fixed-Point Data Types

All structures supported by the Digital Filter block support fixed-point data types. You can specify intermediate fixed-point data types for quantities such as the coefficients, accumulator, and product output for each filter structure. Refer to "Filter Structure Diagrams" on page

10-260 for diagrams depicting the use of these intermediate fixed-point data types in each filter structure.

Dialog Box

Different items appear on the Digital Filter block dialog depending on whether you select **Specify filter characteristics in dialog** or `Specify discretetime filter object (DFILT)` in the **Filter source** group box. Refer to the following sections for details:

- "Specify Filter Characteristics in Dialog" on page 10-245
- "Specify Discrete-Time Filter Object" on page 10-256

Specify Filter Characteristics in Dialog

The **Main** pane of the Digital Filter block dialog appears as follows when **Specify filter characteristics in dialog** is specified in the **Filter source** group box:
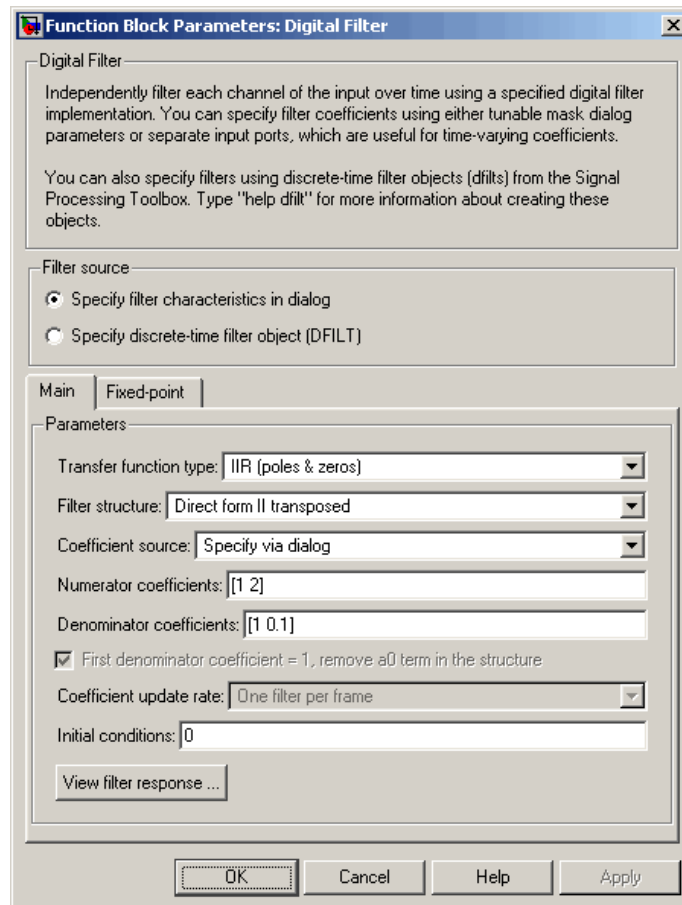
Transfer function type

Select the type of transfer function of the filter; IIR (poles & zeros), IIR (all poles), or FIR (all zeros). Refer to "Supported Filter Structures" on page 10-238 for more information.

Filter structure

> Select the filter structure. The selection of available structures varies depending the setting of the **Transfer function type** parameter. Refer to "Supported Filter Structures" on page 10-238 for more information.

Coefficient source

> Choose how you will specify filter coefficients: via dialog parameters or through input ports. To specify a static filter, choose `Specify via dialog`. To specify a time-varying filter, choose `Input port(s)`. When you select `Input ports(s)`, filter coefficients must come in through block ports. For more information, see "Specifying Static Filters" on page 3-10 and "Specifying Time-Varying Filters" on page 3-11.

Numerator coefficients

> Specify the vector of numerator coefficients of the filter's transfer function.
>
> This parameter is only visible when the **Coefficient source** parameter is set to `Specify via dialog` *and* when the selected filter structure lends itself to specification with numerator coefficients. Tunable.

Denominator coefficients

> Specify the vector of denominator coefficients of the filter's transfer function.
>
> This parameter is only visible when the **Coefficient source** parameter is set to `Specify via dialog` *and* when the selected filter structure lends itself to specification with denominator coefficients. Tunable.

Reflection coefficients

> Specify the vector of reflection coefficients of the filter's transfer function.
>
> This parameter is only visible when the **Coefficient source** parameter is set to `Specify via dialog` *and* when the selected

filter structure lends itself to specification with reflection coefficients. Tunable.

SOS matrix (Mx6)

Specify an *M*-by-6 *SOS matrix* containing coefficients of a second-order section (SOS) filter, where *M* is the number of sections. You can use the `ss2sos` and `tf2sos` functions from the Signal Processing Toolbox to check whether your SOS matrix is valid. For more on the requirements of the SOS matrix, see "Specifying the SOS Matrix (Biquadratic Filter Coefficients)" on page 3-16.

This parameter is only visible when the selected filter structure is biquadratic. Tunable.

Scale values

Specify the scale values to be applied before and after each section of a biquadratic filter.

- If you specify a scalar, that value is applied before the first filter section. The rest of the scale values are set to 1.

- You can also specify a vector with *M* + 1 elements, assigning a different value to each scale. Refer to "Filter Structure Diagrams" on page 10-260 for diagrams depicting the use of scale values in biquadratic filter structures.

This parameter is only visible when the selected filter structure is biquadratic. Tunable.

First denominator coefficient = 1, remove a0 term in the structure

Select this parameter to reduce the number of computations the block must make to produce the output by omitting the $1 / a_0$ term in the filter structure. 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.

This parameter is only enabled when the **Coefficient source** parameter is set to Input port(s) *and* when the selected filter structure lends itself to this specification. See "Removing the a0 Term in the Filter Structure" on page 3-15 for a diagram and details.

Coefficient update rate

Specify how often the block updates time-varying filters; once per sample or once per frame. This parameter only affects the output when the input signal is frame based.

This parameter is only enabled when the **Coefficient source** parameter is set to Input port(s). For more information, see "Specifying Time-Varying Filters" on page 3-11.

Initial conditions

Specify the initial conditions of the filter states. To learn how to specify initial conditions, see "Specifying Initial Conditions" on page 10-241.

Initial conditions on zeros side

(Not shown in dialog above). Specify the initial conditions for the filter states on the side of the filter structure with the zeros (b_0, b_1, b_2, ...); see the diagram below.

This parameter is enabled only when the filter has both poles and zeros, *and* when you select a structure such as direct form I, which has separate filter states corresponding to the poles (a_k) and zeros (b_k). To learn how to specify initial conditions, see "Specifying Initial Conditions" on page 10-241.

Initial conditions on poles side

(Not shown in dialog above). Specify the initial conditions for the filter states on the side of the filter structure with the poles (a_0, a_1, a_2, ...); see the diagram below.

This parameter is enabled only when the filter has both poles and zeros, *and* when you select a structure such as direct form I, which has separate filter states corresponding to the poles (a_k) and zeros

Digital Filter
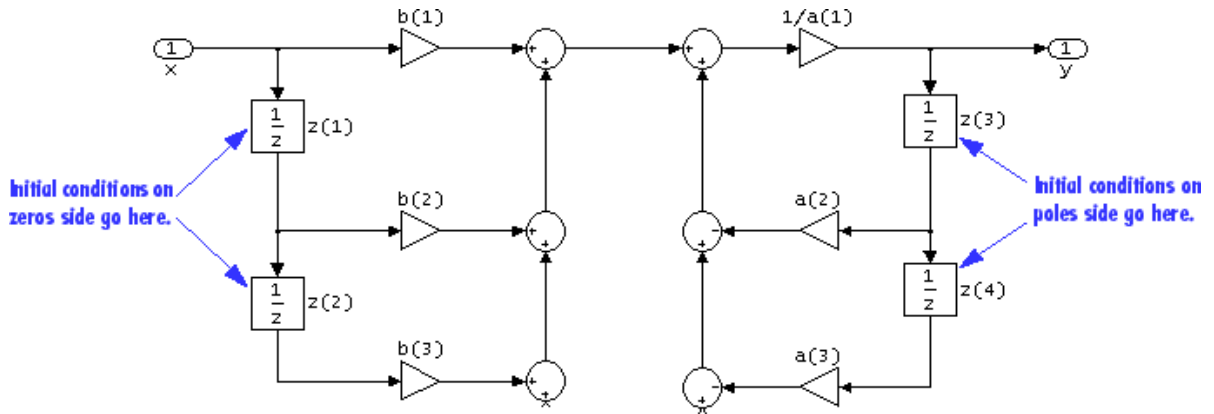
The **Fixed point** pane of the Digital Filter block dialog appears as follows when **Specify filter characteristics in dialog** is specified in the **Filter source** group box:
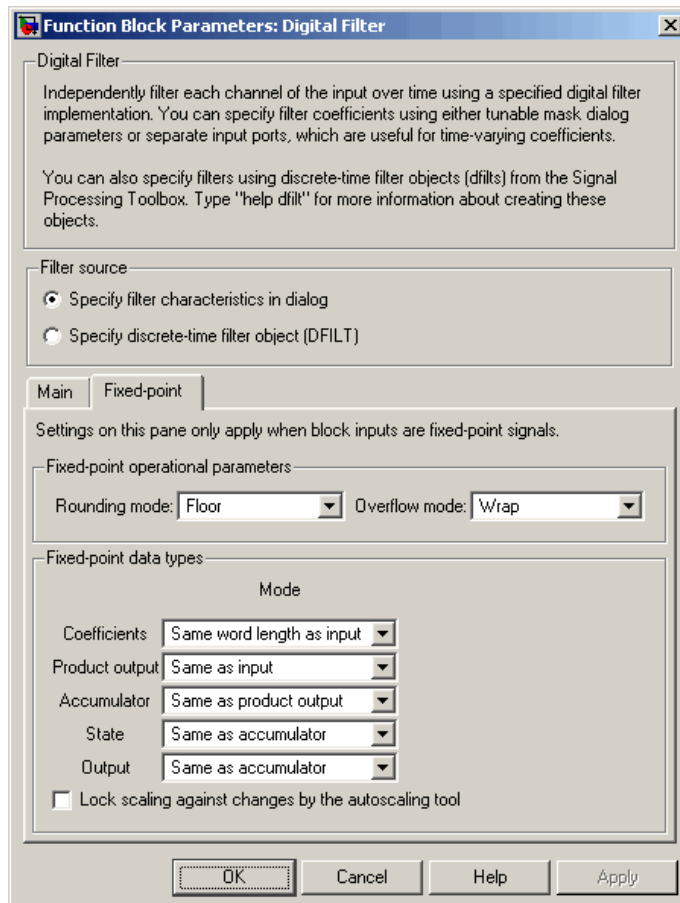
Rounding mode

Select the rounding mode for fixed-point operations. The filter coefficients do not obey this parameter; they always round to `Nearest`.

Overflow mode

Select the overflow mode for fixed-point operations. The filter coefficients do not obey this parameter; they are always saturated.

Section I/O

Choose how you will specify the word length and the fraction length of the fixed-point data type going into and coming out of each section of a biquadratic filter. Refer to "Filter Structure Diagrams" on page 10-260 for illustrations depicting the use of the section I/O data type in this block.

This parameter is only visible when the selected filter structure is biquadratic:

- When you select Same as input, these characteristics will match those of the input to the block.

- When you select Binary point scaling, you are able to enter the word and fraction lengths of the section input and output, in bits.

- When you select Slope and bias scaling, you are able to enter the word lengths, in bits, and the slopes of the section input and output. This block requires power-of-two slope and a bias of zero.

Tap sum

Choose how you will specify the word length and the fraction length of the tap sum data type of a direct form symmetric or direct form antisymmetric filter. Refer to "Filter Structure Diagrams" on page 10-260 for illustrations depicting the use of the tap sum data type in this block.

This parameter is only visible when the selected filter structure is either Direct form symmetric or Direct form antisymmetric:

- When you select Same as input, these characteristics will match those of the input to the block.

- When you select Binary point scaling, you are able to enter the word length and the fraction length of the tap sum accumulator, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the tap sum accumulator. This block requires power-of-two slope and a bias of zero.

Multiplicand

Choose how you will specify the word length and the fraction length of the multiplicand data type of a direct form I transposed or biquadratic direct form I transposed filter. Refer to "Filter Structure Diagrams" on page 10-260 for illustrations depicting the use of the multiplicand data type in this block.

This parameter is only visible when the selected filter structure is either `Direct form I transposed` or `Biquad direct form I transposed (SOS)`:

- When you select `Same as output`, these characteristics will match those of the output to the block.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the multiplicand data type, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the multiplicand data type. This block requires power-of-two slope and a bias of zero.

Coefficients

Choose how you will specify the word length and the fraction length of the filter coefficients (numerator and/or denominator). Refer to "Filter Structure Diagrams" on page 10-260 for illustrations depicting the use of the coefficient data types in this block:

- When you select `Same word length as input`, the word length of the filter coefficients will match that of the input to the block. In this mode, the fraction length of the coefficients is automatically set to the binary-point only scaling that provides

you with the best precision possible given the value and word length of the coefficients.

- When you select `Specify word length`, you are able to enter the word length of the coefficients, in bits. In this mode, the fraction length of the coefficients is automatically set to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the coefficients, in bits. If applicable, you are able to enter separate fraction lengths for the numerator and denominator coefficients.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the coefficients. If applicable, you are able to enter separate slopes for the numerator and denominator coefficients. This block requires power-of-two slope and a bias of zero.

- The filter coefficients do not obey the **Rounding mode** and the **Overflow mode** parameters; they are always saturated and rounded to `Nearest`.

Product output

Use this parameter to specify how you would like to designate the product output word and fraction lengths. Refer to "Filter Structure Diagrams" on page 10-260 and "Multiplication Data Types" on page 8-16 for illustrations depicting the use of the product output data type in this block:

- When you select `Same as input`, these characteristics will match those of the input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the product output, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the product

output. This block requires power-of-two slope and a bias of zero.

Accumulator

Use this parameter to specify how you would like to designate the accumulator word and fraction lengths. Refer to "Filter Structure Diagrams" on page 10-260 and "Multiplication Data Types" on page 8-16 for illustrations depicting the use of the accumulator data type in this block:

- When you select `Same as input`, these characteristics will match those of the input to the block.

- When you select `Same as product output`, these characteristics will match those of the product output.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the accumulator, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the accumulator. This block requires power-of-two slope and a bias of zero.

State

Use this parameter to specify how you would like to designate the state word and fraction lengths. Refer to "Filter Structure Diagrams" on page 10-260 for illustrations depicting the use of the state data type in this block.

This parameter is not visible for direct form and direct form I filter structures.

- When you select `Same as input`, these characteristics will match those of the input to the block.

- When you select `Same as accumulator`, these characteristics will match those of the accumulator.

Digital Filter

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the accumulator, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the accumulator. This block requires power-of-two slope and a bias of zero.

Output

Choose how you will specify the output word length and fraction length:

- When you select `Same as input`, these characteristics will match those of the input to the block.

- When you select `Same as accumulator`, these characteristics will match those of the accumulator.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the output, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the output. This block requires power-of-two slope and a bias of zero.

Lock scaling against changes by the autoscaling tool

Select this parameter to prevent any fixed-point scaling you specify in this block mask from being overriden by the autoscaling tool in the Fixed-Point Settings interface. For more information about the autoscaling tool, refer to "Fixed-Point Settings Interface" on page 8-28.

Specify Discrete-Time Filter Object

The **Main** pane of the Digital Filter block dialog appears as follows when **Specify discrete-time filter object (DFILT)** is specified in the **Filter source** group box:
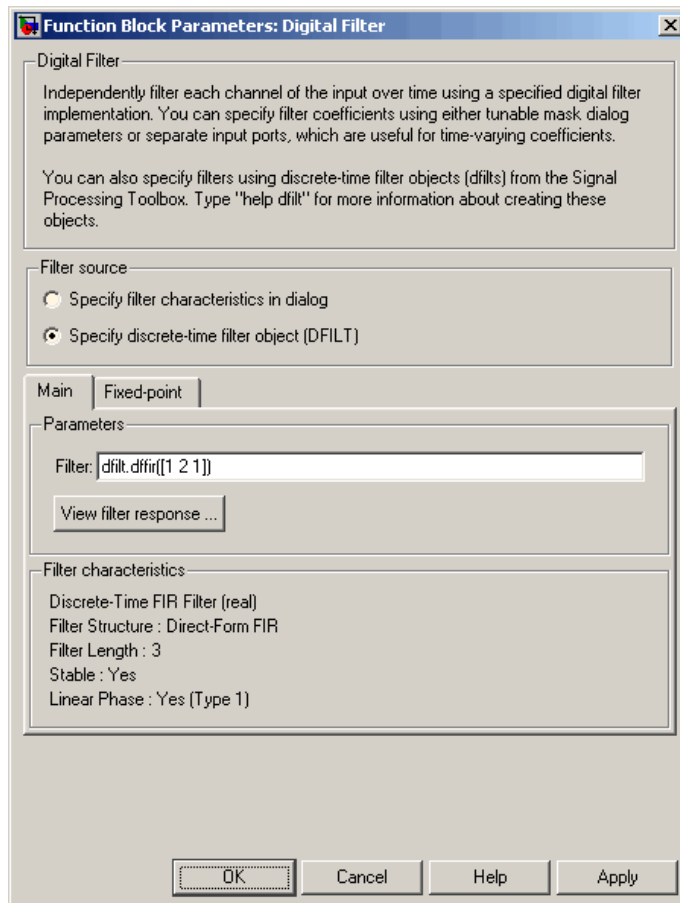
Filter

Specify the discrete-time filter object (dfilt) that you would like the block to implement. You can do this in one of three ways:

- You can fully specify the dfilt object in the block mask, as shown in the default value.

Digital Filter

- You can enter the variable name of a `dfilt` object that is defined in the MATLAB or model workspace.

- You can enter a variable name for a `dfilt` object that is not yet defined.

  For more information on creating `dfilt` objects, refer to the `dfilt` function reference page in the Signal Processing Toolbox documentation.

View filter response

This button opens the Filter Visualization Tool (`fvtool`) from the Signal Processing Toolbox and displays the filter response of the `dfilt` object specified in the **Filter** parameter. For more information on FVTool, refer to the Signal Processing Toolbox documentation.

Characteristics of the `dfilt` object specified in the **Filter** parameter are displayed on this pane in the **Filter characteristics** group box. You cannot change these characteristics directly on the block mask. Instead, you must edit the filter object directly. Refer to the `dfilt` function reference page in the Signal Processing Toolbox documentation.

The **Fixed-point** pane of the Digital Filter block dialog appears as follows when **Specify discrete-time filter object (DFILT)** is specified in the **Filter source** group box:

Function Block Parameters: Digital Filter

Digital Filter

Independently filter each channel of the input over time using a specified digital filter implementation. You can specify filter coefficients using either tunable mask dialog parameters or separate input ports, which are useful for time-varying coefficients.

You can also specify filters using discrete-time filter objects (dfilts) from the Signal Processing Toolbox. Type "help dfilt" for more information about creating these objects.

Filter source

○ Specify filter characteristics in dialog

⦿ Specify discrete-time filter object (DFILT)

| Main | Fixed-point |

Settings on this pane only apply when block inputs are fixed-point signals.

Fixed-point operational parameters

Rounding mode: Floor   Overflow mode: Wrap

Fixed-point data types

                      Mode
 Coefficients   Same word length as input
Product output   Same as input
 Accumulator   Same as product output
   Output   Same as accumulator

OK   Cancel   Help   Apply

The fixed-point settings of the filter object specified on the **Main** pane are displayed on the **Fixed-point** pane. You cannot change these settings directly on the block mask. To change the fixed-point settings you must edit the filter object directly.

For more information on discrete-time filter objects, refer to the dfilt function reference page in the Signal Processing Toolbox documentation.

Digital Filter

Filter Structure Diagrams

The diagrams in the following sections show the filter structures supported by the Digital Filter block. They also show the data types used in the filter structures for fixed-point signals. You can set the coefficient, output, accumulator, product output, and state data types shown in these diagrams in the block dialog. This is discussed in "Dialog Box" on page 10-245.

IIR direct form I
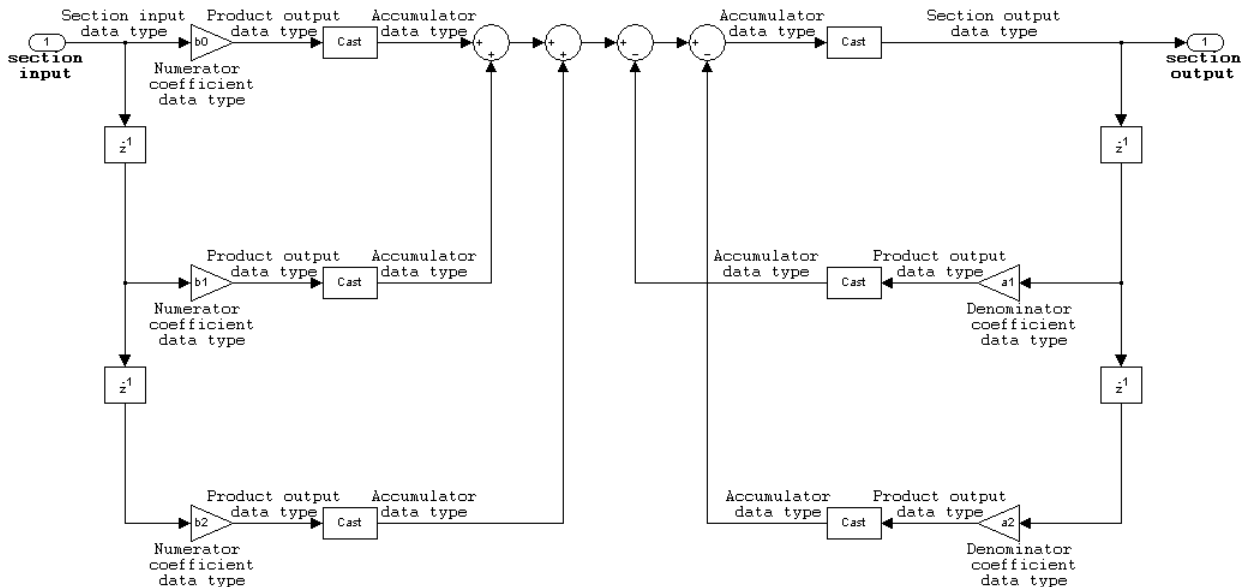

The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs can be real or complex.

- Numerator and denominator coefficients can be real or complex.

- Numerator and denominator coefficients must be the same complexity as each other.

  - When the numerator and denominator coefficients are specified via input ports and have different complexities from each other, you get an error.

  - When the numerator and denominator coefficients are specified in the dialog and have different complexities from each other, the block does not error, but instead processes the filter as if two sets of complex coefficients are provided. The coefficient set that is

real-valued is treated as if it is a complex vector with zero-valued imaginary parts.

- Numerator and denominator coefficients must have the same word length. They can have different fraction lengths.

- The State data type cannot be specified on the block mask for this structure, because the input and output states have the same data types as the input and output buffers.

IIR direct form I transposed

The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs can be real or complex.

- Numerator and denominator coefficients can be real or complex.

- Numerator and denominator coefficients must be the same complexity as each other.

  - When the numerator and denominator coefficients are specified via input ports and have different complexities from each other, you get an error.

- When the numerator and denominator coefficients are specified in the dialog and have different complexities from each other, the block does not error, but instead processes the filter as if two sets of complex coefficients are provided. The coefficient set that is real-valued is treated as if it is a complex vector with zero-valued imaginary parts.

- States are complex when either the input or the coefficients are complex.

- Numerator and denominator coefficients must have the same word length. They can have different fraction lengths.
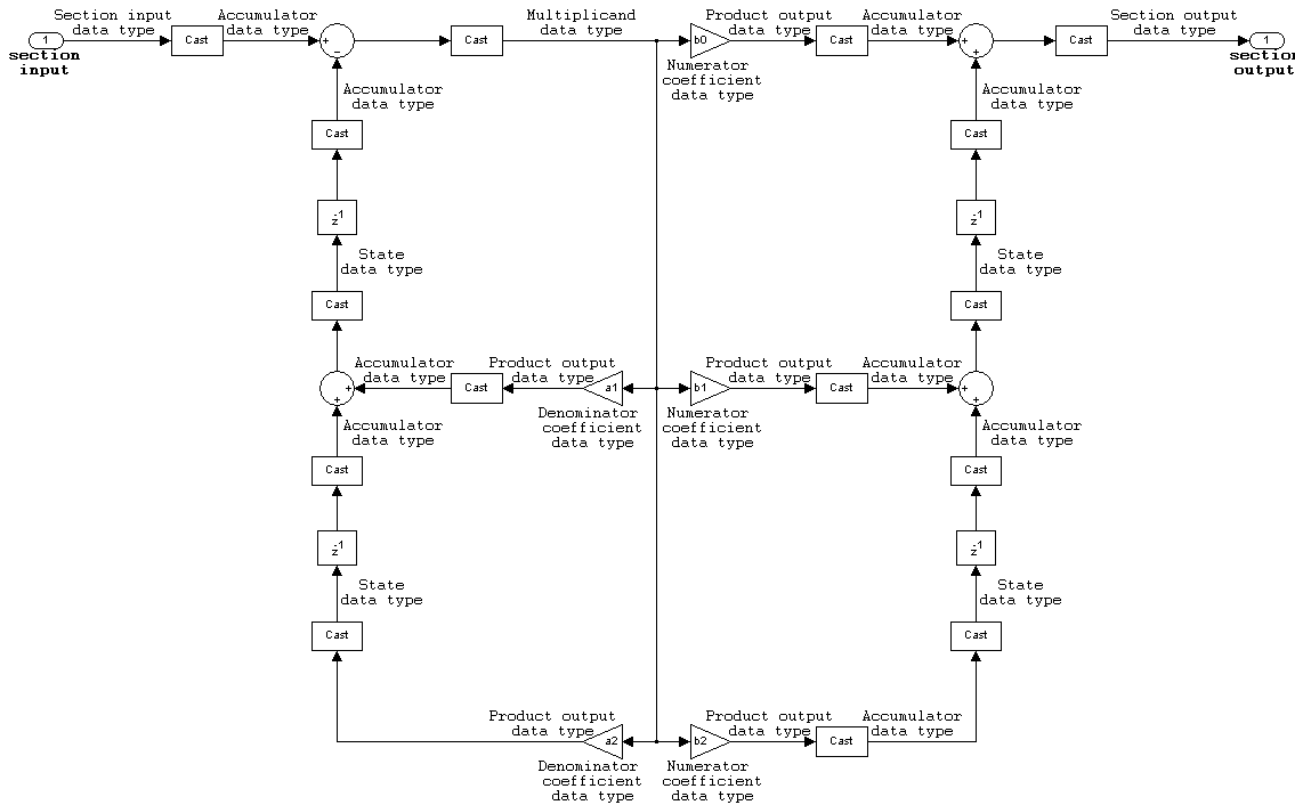
IIR direct form II

The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs can be real or complex.

- Numerator and denominator coefficients can be real or complex.

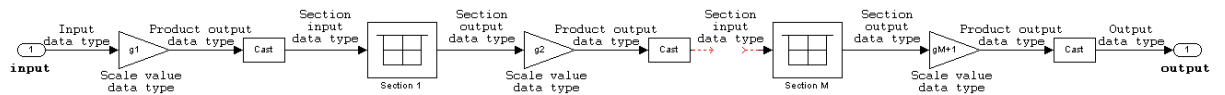- Numerator and denominator coefficients must be the same complexity as each other.

  - When the numerator and denominator coefficients are specified via input ports and have different complexities from each other, you get an error.

  - When the numerator and denominator coefficients are specified in the dialog and have different complexities from each other, the block does not error, but instead processes the filter as if two sets of complex coefficients are provided. The coefficient set that is

real-valued is treated as if it is a complex vector with zero-valued imaginary parts.

- States are complex when either the inputs or the coefficients are complex.

- Numerator and denominator coefficients must have the same word length. They can have different fraction lengths.
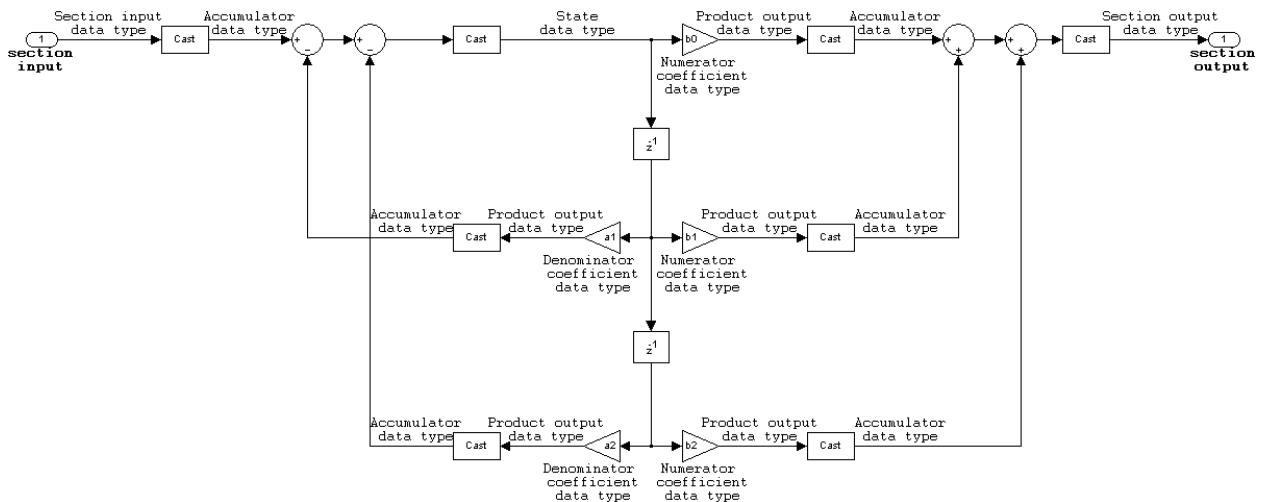
IIR direct form II transposed

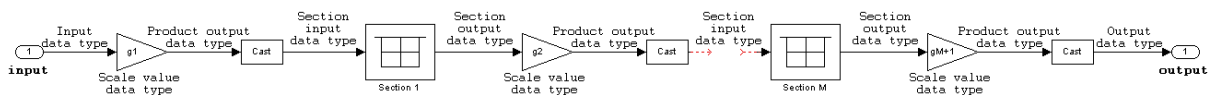The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs can be real or complex.

- Numerator and denominator coefficients can be real or complex.

- Numerator and denominator coefficients must be the same complexity as each other.

  - When the numerator and denominator coefficients are specified via input ports and have different complexities from each other, you get an error.

- When the numerator and denominator coefficients are specified in the dialog and have different complexities from each other, the block does not error, but instead processes the filter as if two sets of complex coefficients are provided. The coefficient set that is real-valued is treated as if it is a complex vector with zero-valued imaginary parts.

- States are complex when either the inputs or the coefficients are complex.

- Numerator and denominator coefficients must have the same word length. They can have different fraction lengths.

Digital Filter
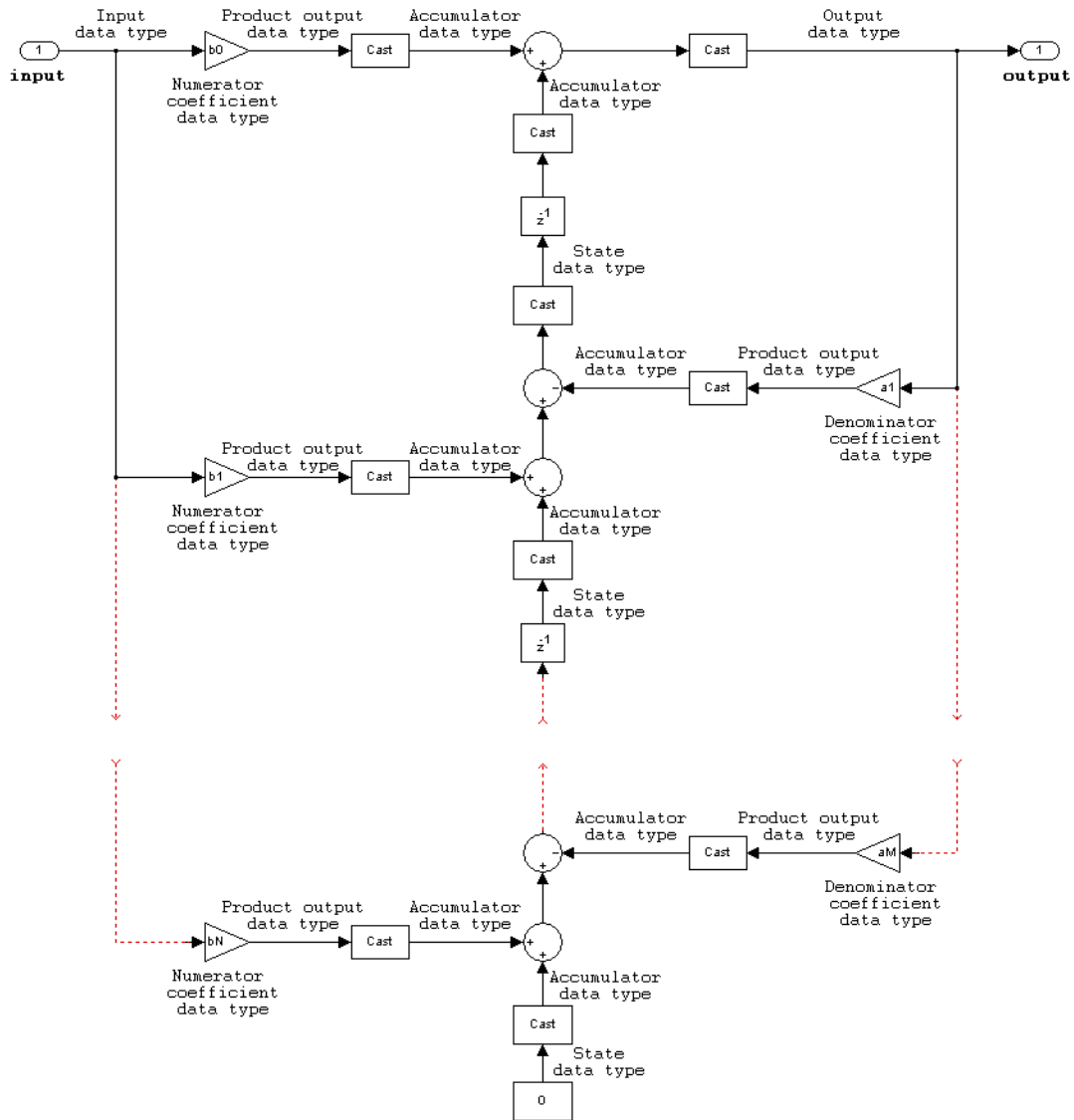
IIR biquadratic direct form I
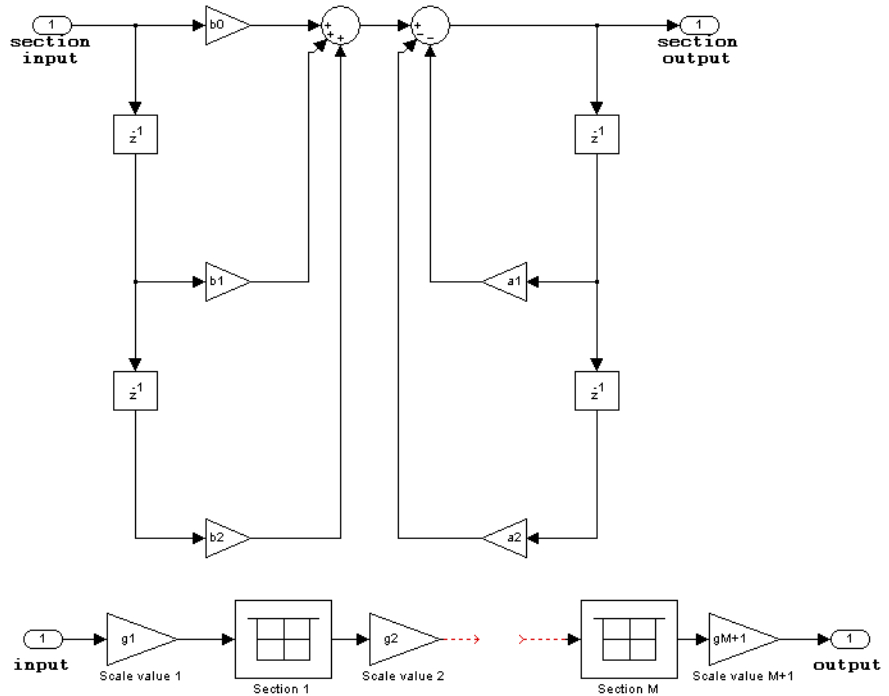
The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs and coefficients can be real or complex.

- Numerator and denominator coefficients can be real or complex.

- Specify the coefficients by a *M*-by-6 matrix in the block mask. You cannot specify coefficients by input ports for this filter structure.

- When the a0 element of any row is not equal to one, that row is normalized by a0 prior to filtering.

- States are complex when either the inputs or the coefficients are complex.

- You cannot specify the state data type on the block mask for this structure, because the input and output states have the same data types as the input.

- Scale values must have the same complexity as the coefficient SOS matrix.

- The scale value parameter must be a scalar or a vector of length $M+1$, where M is the number of sections.

- The **Section I/O** parameter determines the data type for the section input and output data types. The section input and stage output data type must have the same word length but can have different fraction lengths.

  The following diagram shows the data types for one section of the filter.

The following diagram shows the data types between filter sections.
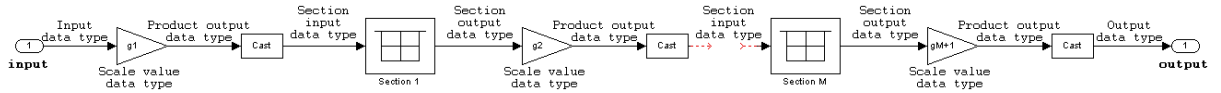
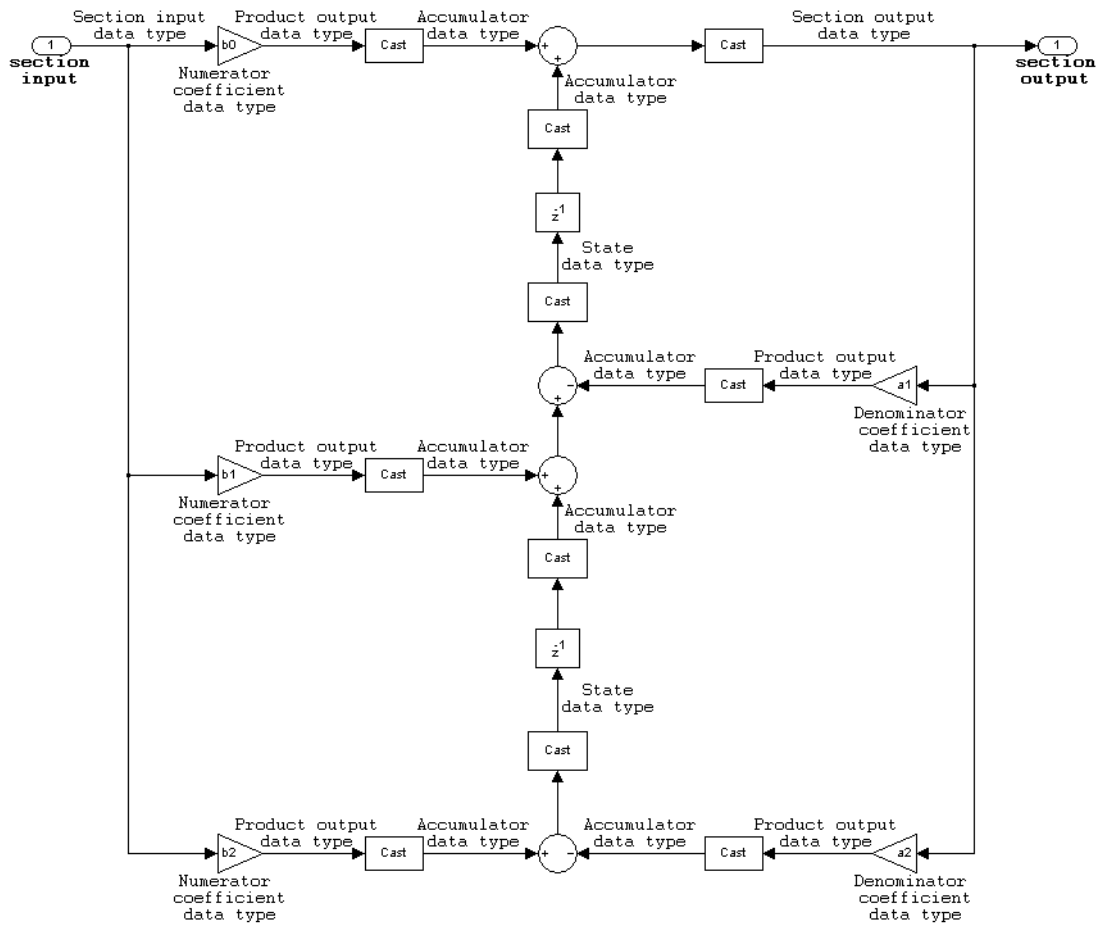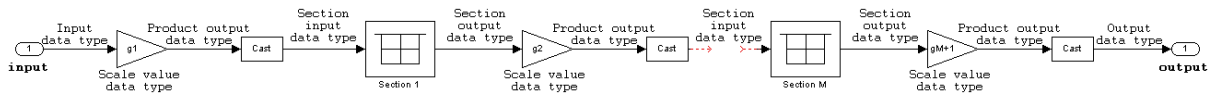**IIR biquadratic direct form I transposed**
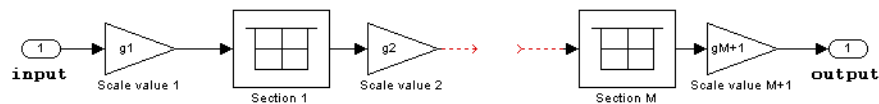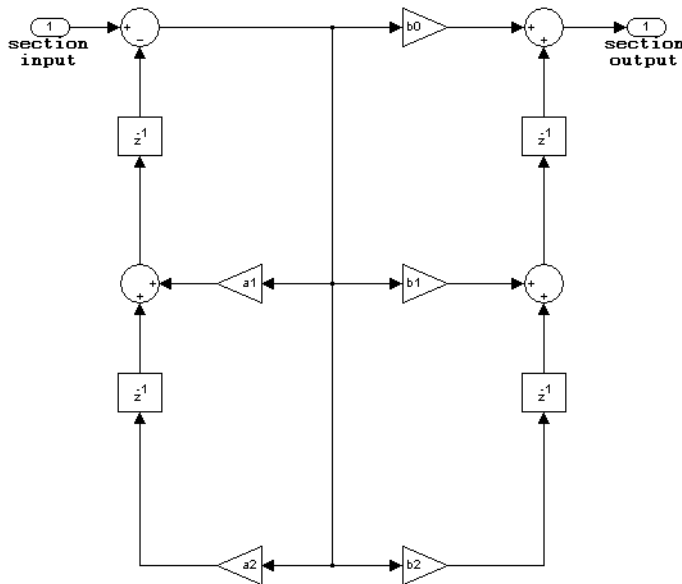
The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs and coefficients can be real or complex.

- Numerator and denominator coefficients can be real or complex.

Digital Filter

- Specify the coefficients by a *M*-by-6 matrix in the block mask. You cannot specify coefficients by input ports for this filter structure.

- When the a0 element of any row is not equal to one, that row is normalized by a0 prior to filtering.

- States are complex when either the inputs or the coefficients are complex.

- Scale values must have the same complexity as the coefficient SOS matrix.

- The scale value parameter must be a scalar or a vector of length *M*+1, where *M* is the number of sections.

- The **Section I/O** parameter determines the data type for the section input and output data types. The section input and section output data type must have the same word length but can have different fraction lengths.

The following diagram shows the data types for one section of the filter.

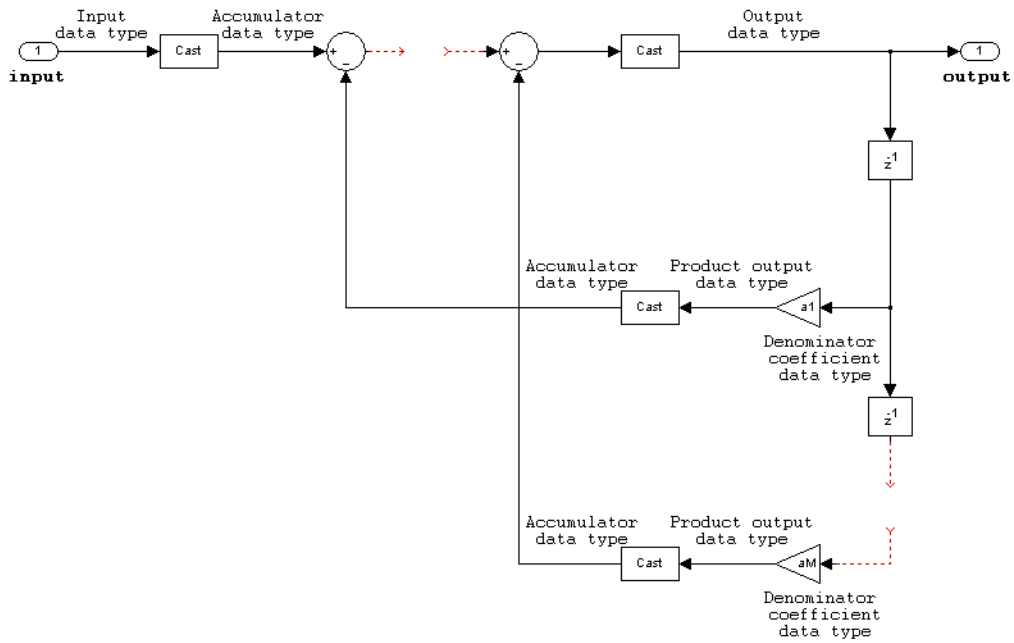The following diagram shows the data types between filter sections.
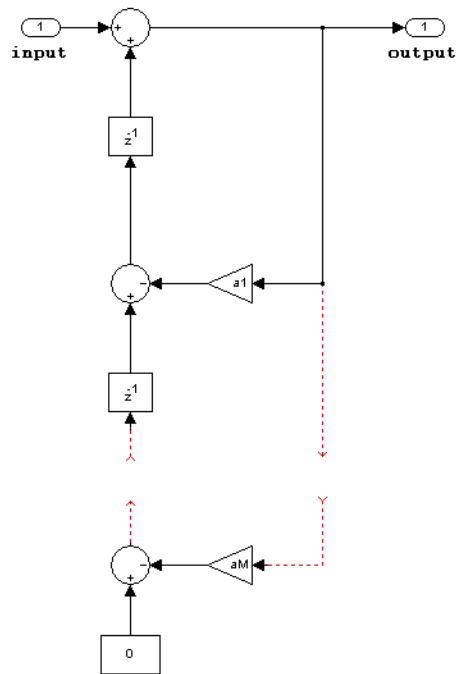
IIR biquadratic direct form II
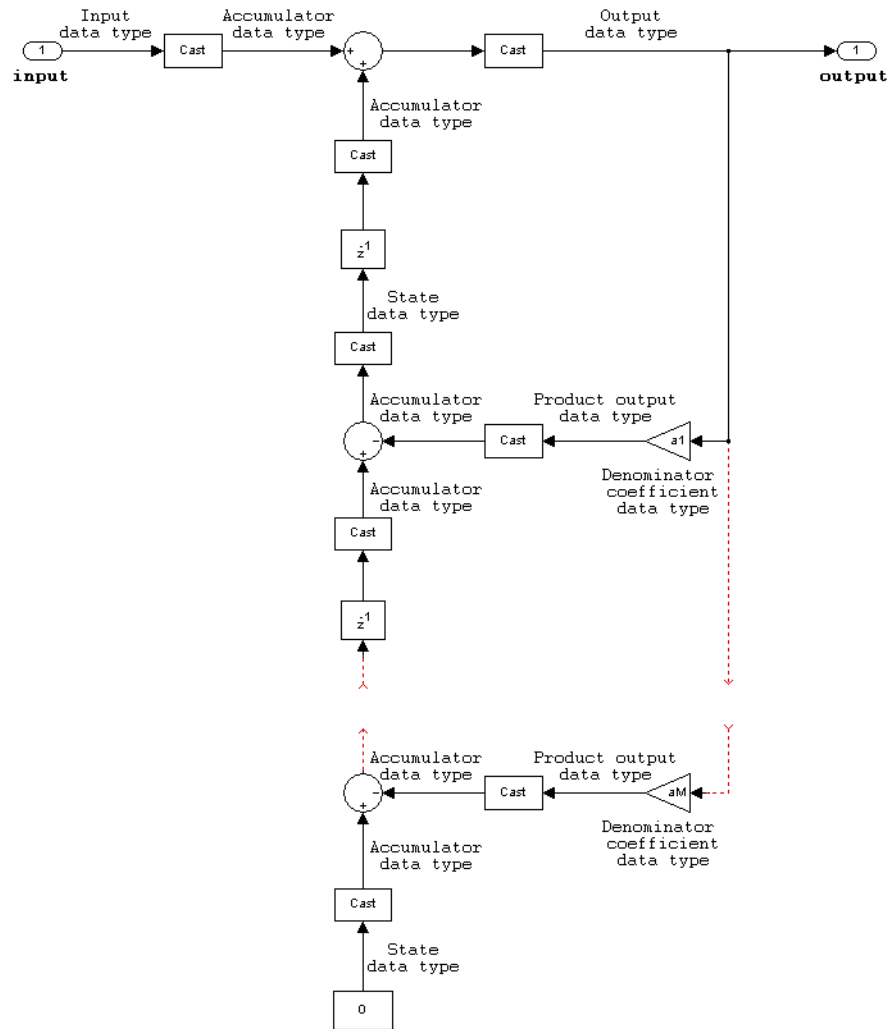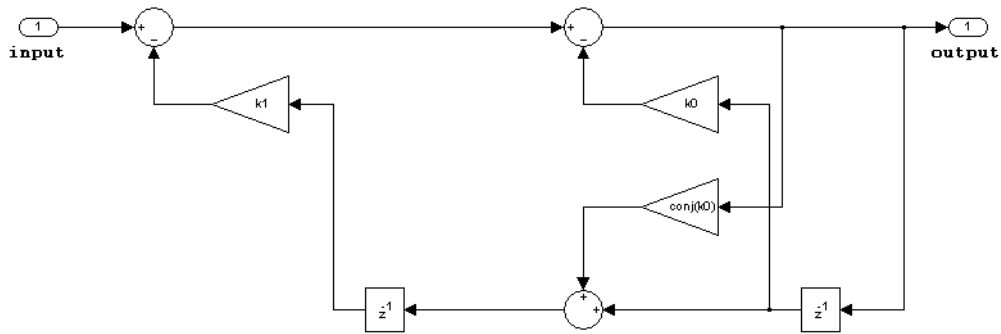

The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs and coefficients can be real or complex.
- Numerator and denominator coefficients can be real or complex.
- Specify the coefficients by a *M*-by-6 matrix in the block mask. You cannot specify coefficients by input ports for this filter structure.
- When the a0 element of any row is not equal to one, that row is normalized by a0 prior to filtering.

- States are complex when either the inputs or the coefficients are complex.

- Scale values must have the same complexity as the coefficient SOS matrix.

- The scale value parameter must be a scalar or a vector of length $M+1$, where M is the number of sections.

- The **Section I/O** parameter determines the data type for the section input and output data types. The section input and section output data type must have the same word length but can have different fraction lengths.

The following diagram shows the data types for one section of the filter.

The following diagram shows the data types between filter sections.

IIR biquadratic direct form II transposed

The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs and coefficients can be real or complex.

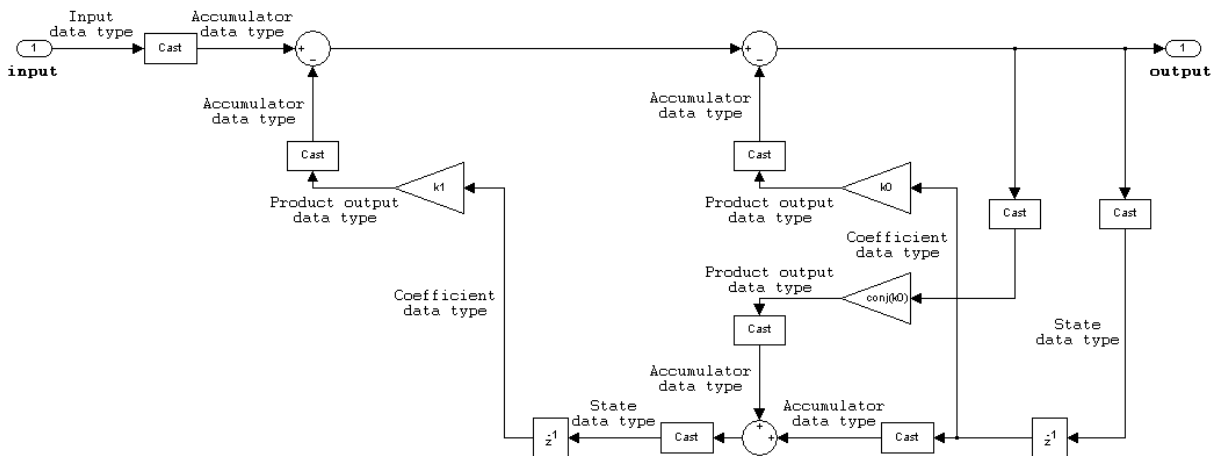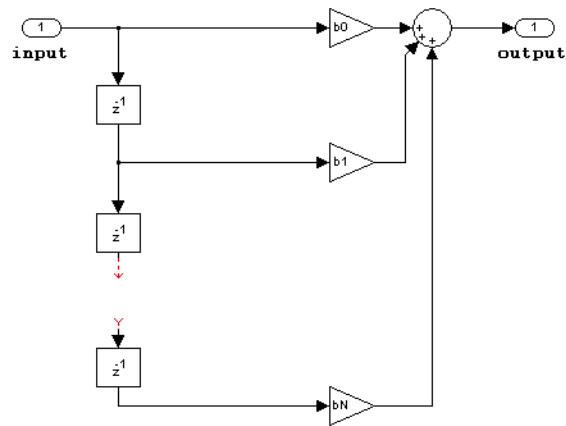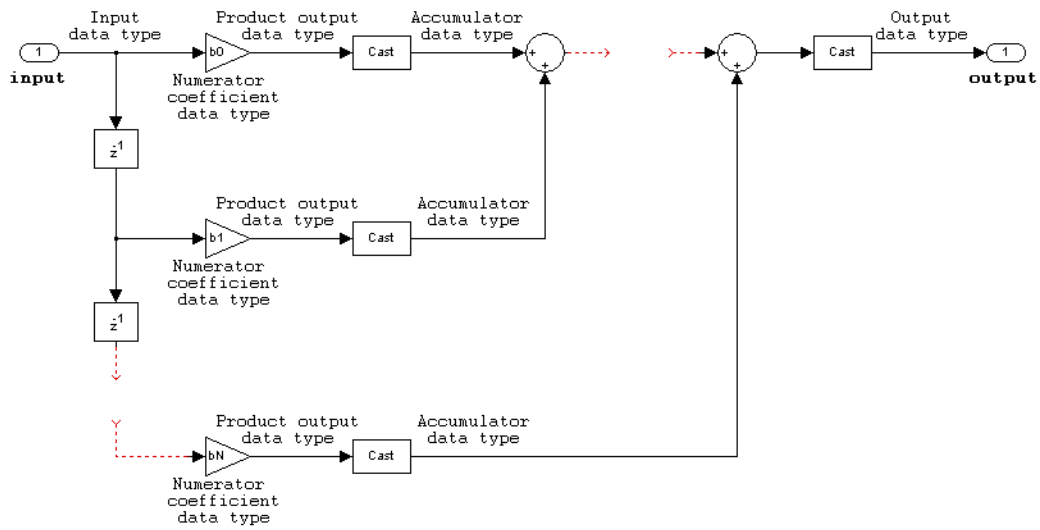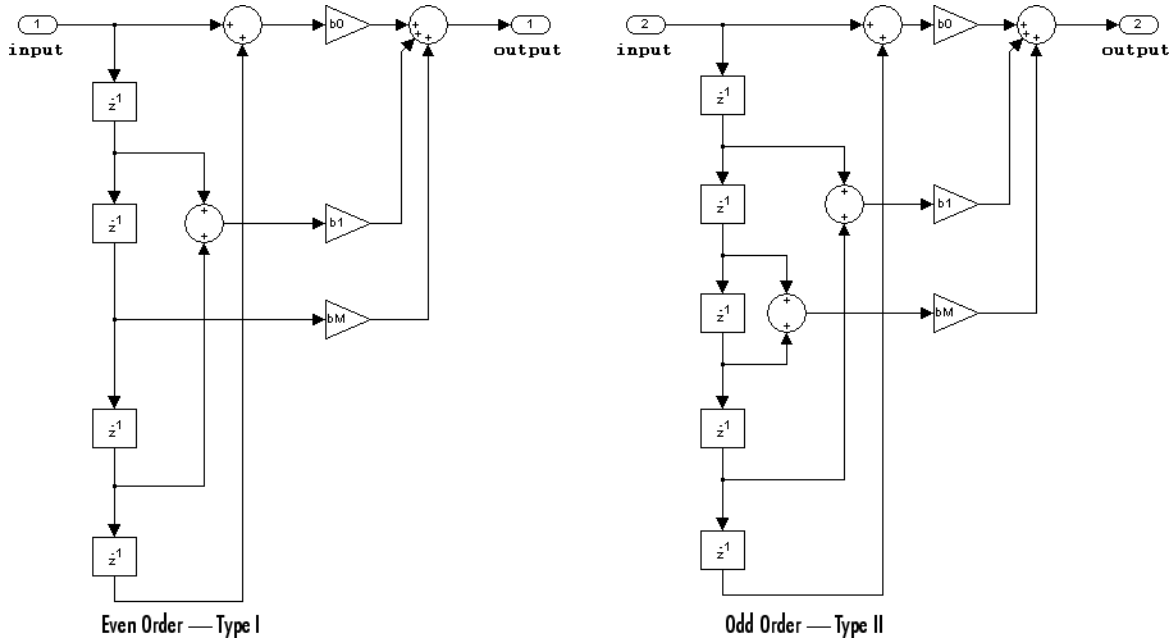- Numerator and denominator coefficients can be real or complex.

- Specify the coefficients by a *M*-by-6 matrix in the block mask. You cannot specify coefficients by input ports for this filter structure.

- When the a0 element of any row is not equal to one, that row is normalized by a0 prior to filtering.

- States are complex when either the inputs or the coefficients are complex.

- Scale values must have the same complexity as the coefficient SOS matrix.

- The scale value parameter must be a scalar or a vector of length M+1, where M is the number of sections.

- The **Section I/O** parameter determines the data type for the section input and output data types. The section input and section output data type must have the same word length but can have different fraction lengths.

The following diagram shows the data types for one section of the filter.

Digital Filter

The following diagram shows the data types between filter sections.

IIR (all poles) direct form
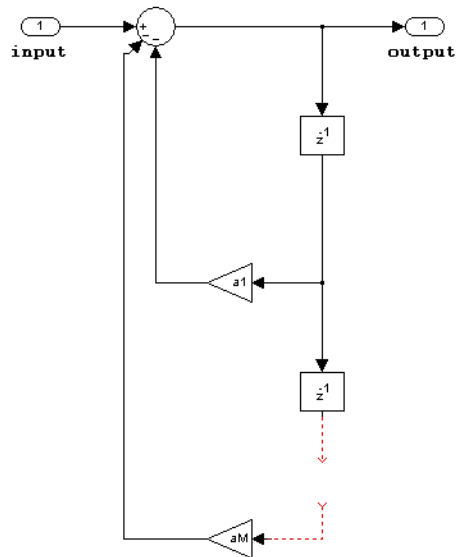
The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs and coefficients can be real or complex.

- Denominator coefficients can be real or complex.

- You cannot specify the state data type on the block mask for this structure, because the input and output states have the same data types as the input.
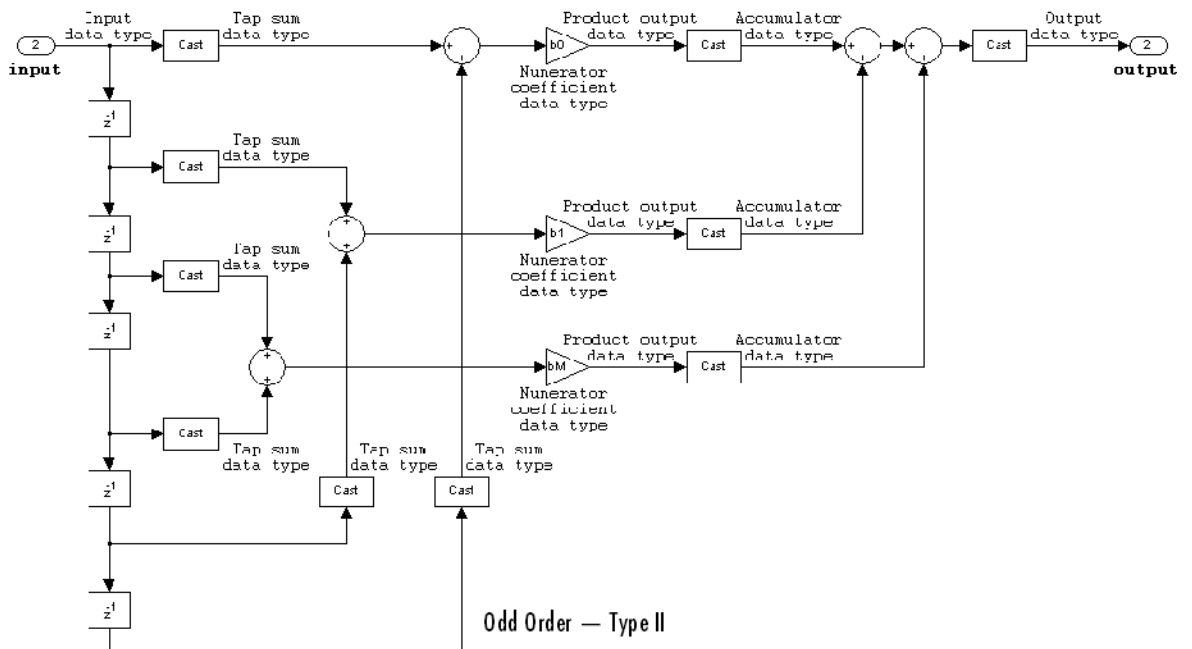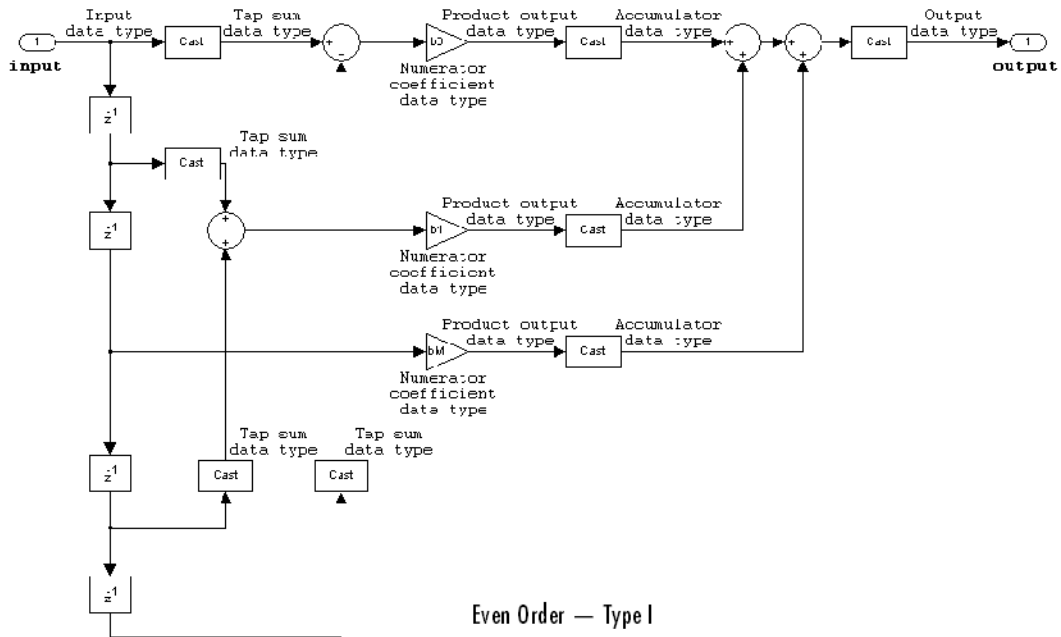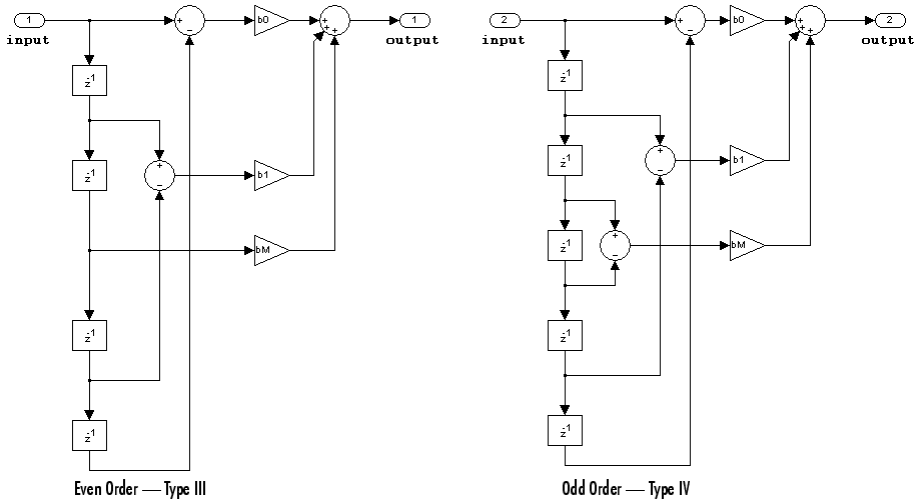
Digital Filter

IIR (all poles) direct form transposed

The following constraints are applicable when processing a fixed-point signal with this filter structure:

• Inputs and coefficients can be real or complex.

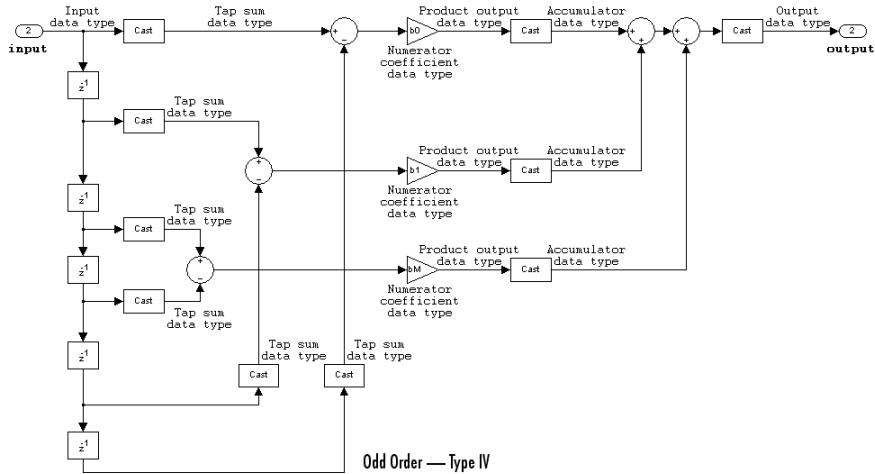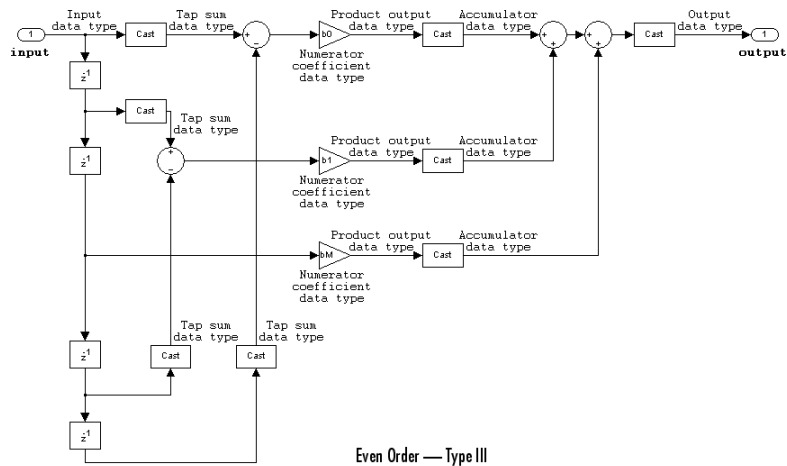• Denominator coefficients can be real or complex.

Digital Filter

IIR (all poles) direct form lattice AR

The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs and coefficients can be real or complex.

- Coefficients can be real or complex.

FIR (all zeros) direct form

The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs can be real or complex.

- Numerator coefficients can be real or complex.

- You cannot specify the state data type on the block mask for this structure, because the input and output states have the same data types as the input.

Digital Filter

FIR (all zeros) direct form symmetric

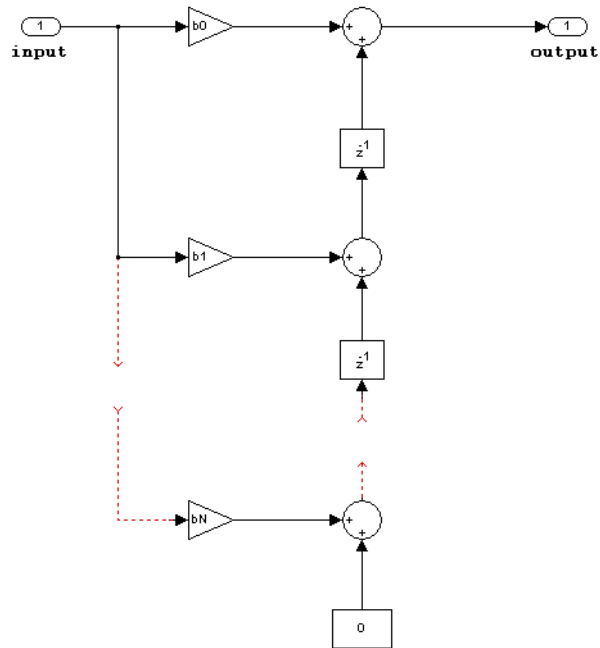Even Order — Type I   Odd Order — Type II

The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs can be real or complex.

- Numerator coefficients can be real or complex.

- You cannot specify the state data type on the block mask for this structure, because the input and output states have the same data types as the input.

- It is assumed that the filter coefficients are symmetric. Only the first half of the coefficients are used for filtering.

- The **Tap Sum** parameter determines the data type the filter uses when it sums the inputs prior to multiplication by the coefficients.

Even Order — Type I

Odd Order — Type II

Digital Filter

FIR (all zeros) direct form antisymmetric

Even Order — Type III

Odd Order — Type IV

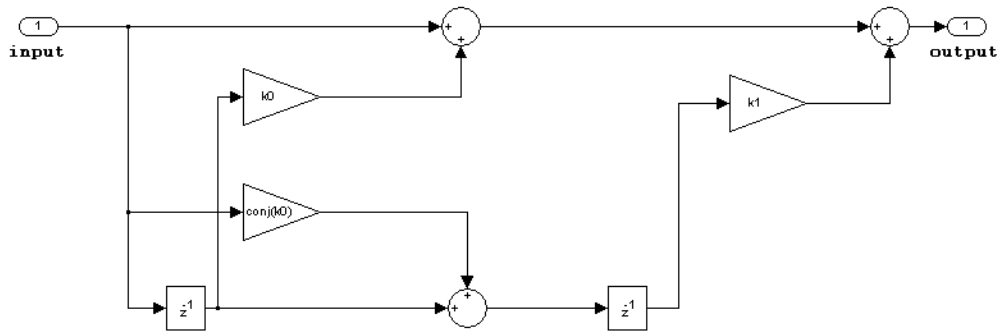The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs can be real or complex.

- Numerator coefficients can be real or complex.

- You cannot specify the state data type on the block mask for this structure, because the input and output states have the same data types as the input.

- It is assumed that the filter coefficients are antisymmetric. Only the first half of the coefficients are used for filtering.

- The **Tap Sum** parameter determines the data type the filter uses when it sums the inputs prior to multiplication by the coefficients.

Even Order — Type III

Odd Order — Type IV

Digital Filter

FIR (all zeros) direct form transposed

The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs can be real or complex.
- Coefficients can be real or complex.
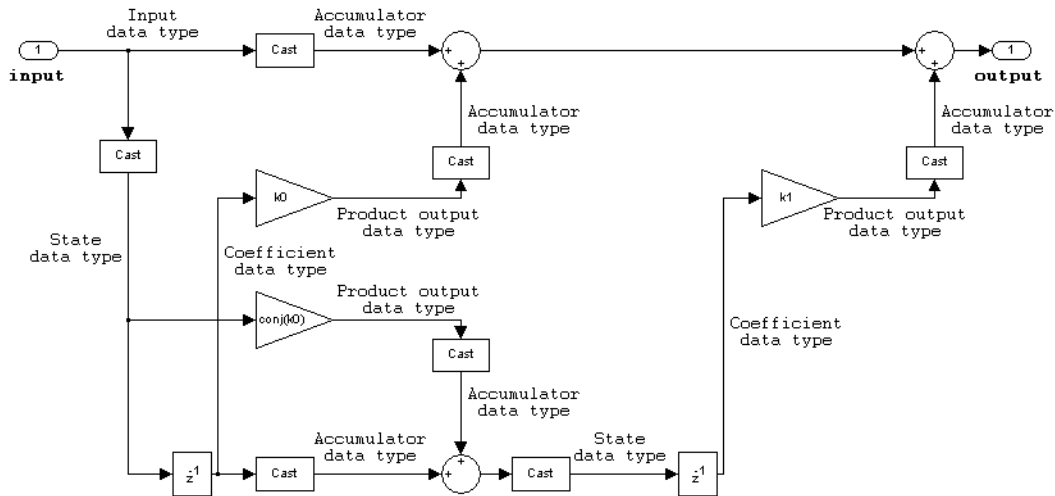- States are complex when either the inputs or the coefficients are complex.
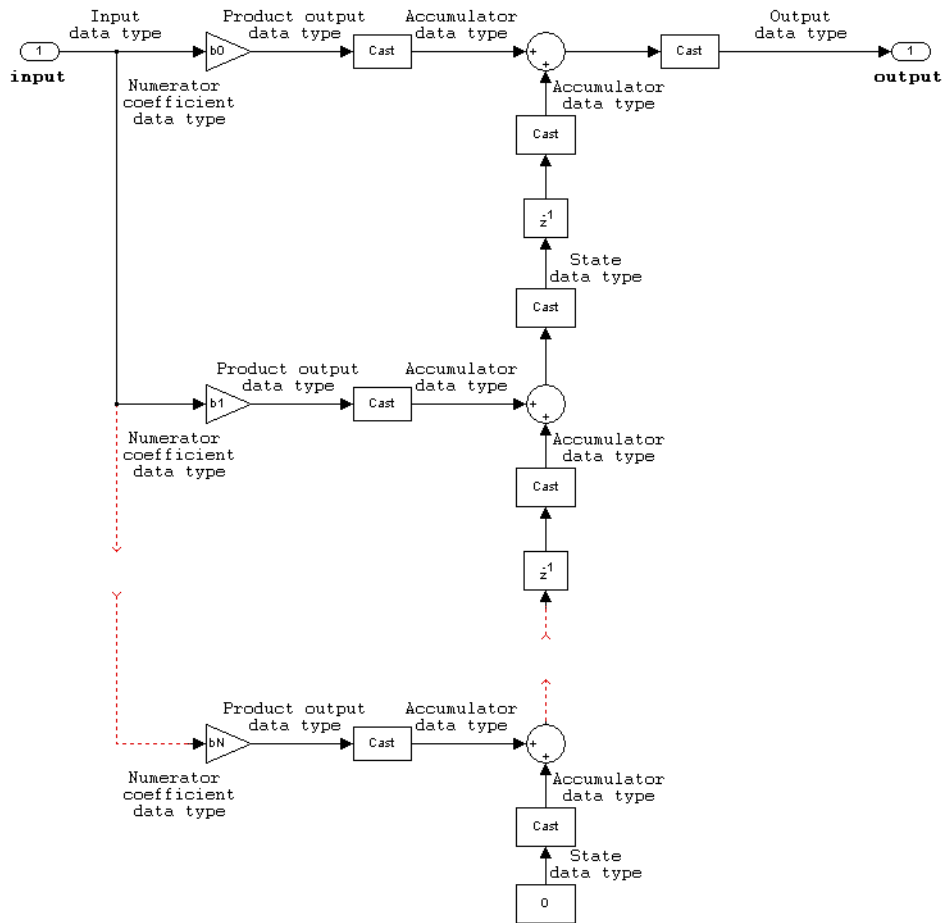
FIR (all zeros) lattice MA

The following constraints are applicable when processing a fixed-point signal with this filter structure:

- Inputs and coefficients can be real or complex.

- Coefficients can be real or complex.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point (signed only)
- 8-, 16-, and 32-bit signed integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Digital Filter Design	Signal Processing Blockset
Filter Realization Wizard	Signal Processing Blockset
dfilt	Signal Processing Toolbox
fdatool	Signal Processing Toolbox
fvtool	Signal Processing Toolbox
sptool	Signal Processing Toolbox

Digital Filter Design

Purpose   Design and implement digital FIR and IIR filters

Library   Filtering / FilterDesigns

Description

FDATool
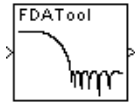
Note Use this block to design, analyze, and then efficiently implement floating-point filters. The following blocks also implement digital filters, but serve slightly different purposes:

- Digital Filter — Use to efficiently implement floating-point or fixed-point filters that you have already designed. This block provides the same exact filter implementation as the Digital Filter Design block.

- Filter Realization Wizard — Use to implement floating-point or fixed-point filters built from Sum, Gain, and Unit Delay blocks. You can either design the filter within this block, or import the coefficients of a filter that you designed elsewhere.

The Digital Filter Design block implements a digital FIR or IIR filter that you design using the Filter Design and Analysis Tool (FDATool) GUI. This block provides the same exact filter implementation as the Digital Filter block.

The block applies the specified filter to each channel of a discrete-time input signal, and outputs the result. The outputs of the block numerically match the outputs of the Digital Filter block, the `filter` function in MATLAB, and the `filter` function in the Filter Design Toolbox.

The sampling frequency, Fs, that you specify in the FDATool GUI should be identical to the sampling frequency of the Digital Filter Design block's input block. When the sampling frequencies of these blocks do not match, the Digital Filter Design block returns a warning message and inherits the sampling frequency of the input block.

Valid Inputs and Corresponding Outputs

The block accepts inputs that are sample-based or frame-based vectors and matrices. The block filters each input channel independently over time, where

- Each *column* of a frame-based vector or matrix is an independent channel.

- Each *element* of a sample-based vector or matrix is an independent channel.

The output has the same dimensions and frame status as the input.

Designing the Filter

Double-click the Digital Filter Design block to open FDATool. Use FDATool to design or import a digital FIR or IIR filter. To learn how to design filters with this block and FDATool, see the following topics:

- "Digital Filter Design Block" on page 3-18

- Topic on the Filter Design and Analysis Tool (FDATool) in the Signal Processing Toolbox documentation.

Tuning the Filter During Simulation

You can tune the filter specifications in FDATool during simulations as long as your changes do not modify the filter length or filter order. The block's filter updates as soon as you apply any filter changes in FDATool.

Digital Filter Design

Dialog Box
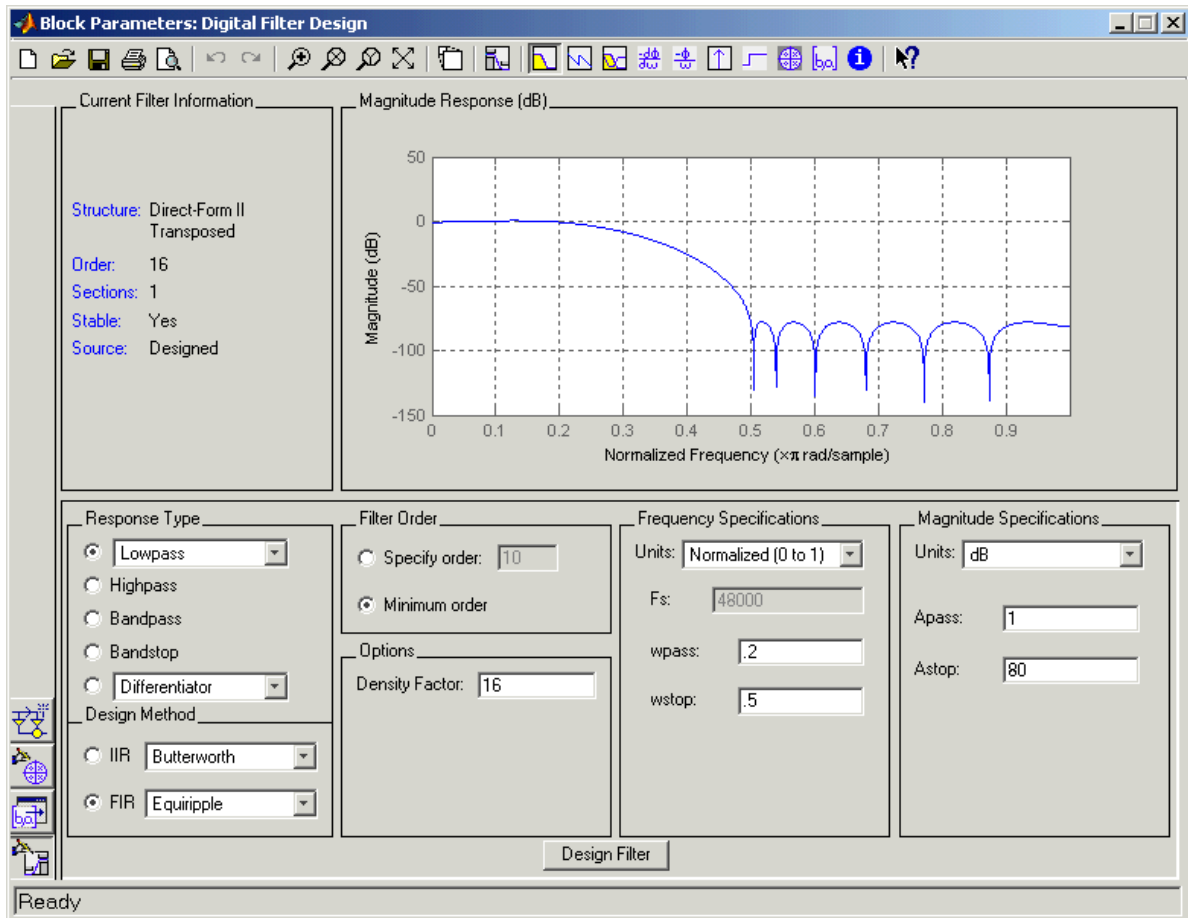
The FDATool GUI Opened from the Digitial Filter Design Block

To get the **Transform Filter** button and the **Set Quantization Parameters** button , install the Filter Design Toolbox. To get the

Targets menu, install the Embedded Target for the TI TMS320C6000™ DSP Platform.

To learn how to use the FDATool GUI, see "Designing the Filter" on page 10-297.

Supported Data Types

- Double-precision floating point
- Single-precision floating point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Analog Filter Design	Signal Processing Blockset
Window Function	Signal Processing Blockset
fdatool	Signal Processing Toolbox
filter	Signal Processing Toolbox
fvtool	Signal Processing Toolbox
sptool	Signal Processing Toolbox
filter	Filter Design Toolbox

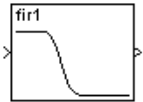To learn how to use this block and FDATool, see the following:

- Chapter 3, "Filters"
- "Digital Filter Design Block" on page 3-18
- Topic on the Filter Design and Analysis Tool (FDATool) in the Signal Processing Toolbox documentation.

Digital FIR Filter Design

Purpose   Design and implement a variety of FIR filters.

Library   Filtering / Filter Designs

Description

Note The Digital FIR Filter Design block is still supported but is likely to be obsoleted in a future release. We strongly recommend replacing this block with the Digital Filter block.

The Digital FIR Filter Design block designs a discrete-time (digital) FIR filter in one of several different band configurations using a window method. Most of these filters are designed using the `fir1` function in the Signal Processing Toolbox, and are real with linear phase response. The block applies the filter to a discrete-time input using the Direct-Form II Transpose Filter block.

An *M*-by-*N* sample-based matrix input is treated as *M*N* independent channels, and an *M*-by-*N* frame-based matrix input is treated as *N* independent channels. In both cases, the block filters each channel independently over time, and the output has the same size and frame status as the input.

For complete details on the classical FIR filter design algorithm, see the description of the `fir1` and `fir2` functions in the Signal Processing Toolbox documentation.

Band Configurations

The band configuration for the filter is set from the **Filter type** pop-up menu. The band configuration parameters below this pop-up menu adapt appropriately to match the **Filter type** selection.

- **Lowpass** and **Highpass**

    In lowpass and highpass configurations, the **Filter order** and **Cutoff frequency** parameters specify the filter design. Frequencies are normalized to half the sample frequency. The figure below shows the frequency response of the default order-22 filter with cutoff at 0.4.
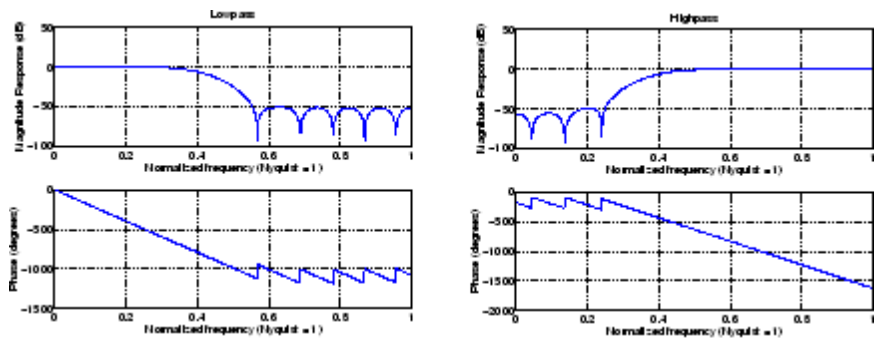
- **Bandpass** and **Bandstop**

  In bandpass and bandstop configurations, the **Filter order**, **Lower cutoff frequency**, and **Upper cutoff frequency** parameters specify the filter design. Frequencies are normalized to half the sample frequency, and the actual filter order is twice the **Filter order** parameter value. The figure below shows the frequency response of the default order-22 filter with lower cutoff at 0.4, and upper cutoff at 0.6.
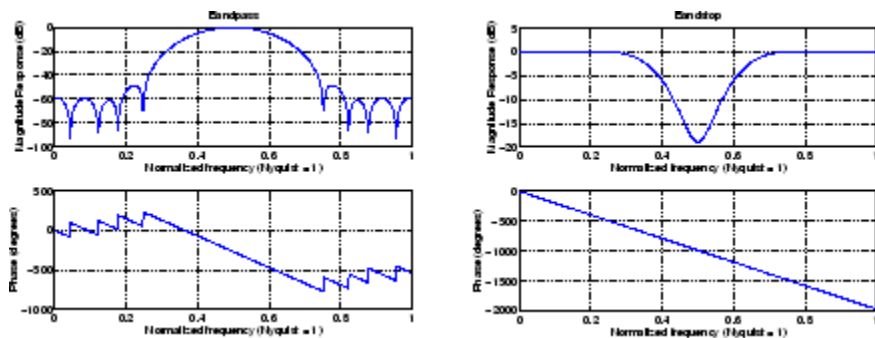

- **Multiband**

  In the multiband configuration, the **Filter order**, **Cutoff frequency vector**, and **Gain in the first band** parameters specify the filter design. The **Cutoff frequency vector** contains frequency points in

the range 0 to 1, where 1 corresponds to half the sample frequency. Frequency points must appear in ascending order. The **Gain in the first band** parameter specifies the gain in the first band: 0 indicates a stopband, and 1 indicates a passband. Additional bands alternate between passband and stopband. The figure below shows the frequency response of the default order-22 filter with five bands, the first a passband.

Cutoff frequency vector = [0.2 0.4 0.6 0.8]
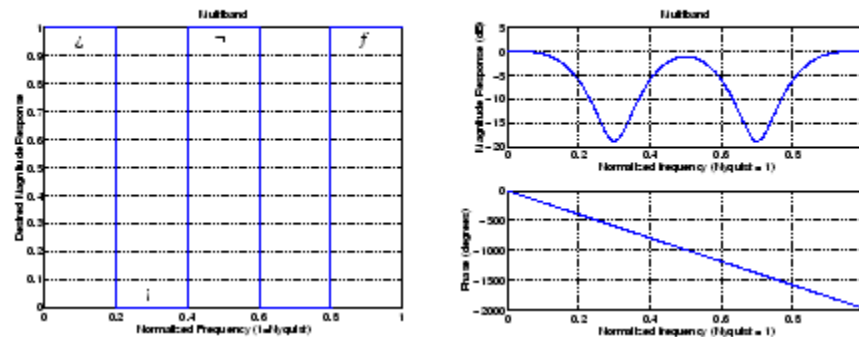

- **Arbitrary shape**

  In the arbitrary shape configuration, the **Filter order**, **Frequency vector**, and **Gains at these frequencies** parameters specify the filter design. The **Frequency vector**, fn, contains frequency points in the range 0 to 1 (inclusive) in ascending order, where 1 corresponds to half the sample frequency. The **Gains at these frequencies** parameter, mn, is a vector containing the desired magnitude response at the corresponding points in the **Frequency vector**. (Note that the specifications for the **Arbitrary shape** configuration are similar to those for the Yule-Walker IIR Filter Design block. Arbitrary-shape filters are designed using the fir2 function in the Signal Processing Toolbox.)

The desired magnitude response of the design can be displayed by typing

```
plot(fn,mn)
```

Duplicate frequencies can be used to specify a step in the response (such as band 2 below). The figure shows an order-100 filter with five bands.
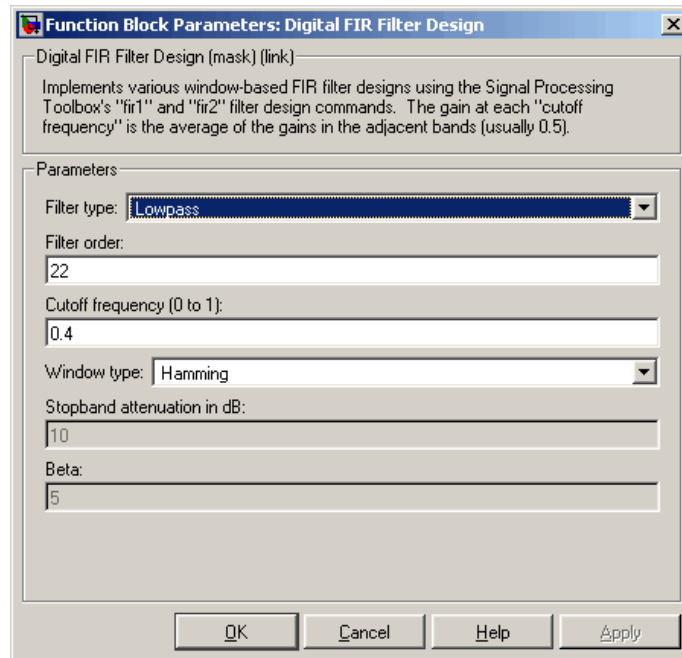
Frequency =   [0.0 0.3 0.3 0.6 0.7 1.0]

Gains =   [0.0 1.0 0.0 0.0 0.5 0.5]

Band: ¿ i ¬ f
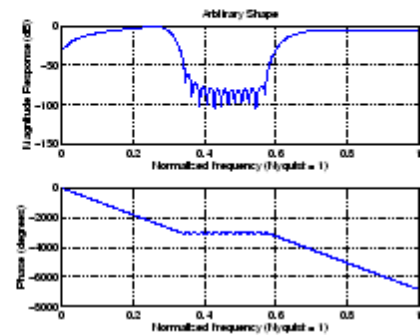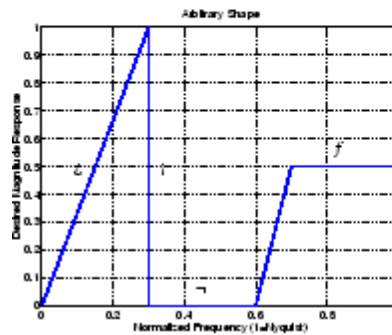
The **Window type** parameter allows you to select from a variety of different windows. See the Window Function block reference for a complete description of the available options.

Digital FIR Filter Design

**Dialog
Box**

The parameters displayed in the dialog box vary for different
design/band combinations. Only some of the parameters listed below
are visible in the dialog box at any one time.

Filter type
> The type of filter to design: **Lowpass**, **Highpass**, **Bandpass**,
> **Bandstop**, **Multiband**, or **Arbitrary Shape**. Tunable.

Filter order
> The order of the filter. The filter length is one more than this
> value. For the **Bandpass** and **Bandstop** configurations, the
> order of the final filter is twice this value.

Cutoff frequency
> The normalized cutoff frequency for the **Highpass** and **Lowpass**
> filter configurations. A value of 1 specifies half the sample
> frequency. Tunable.

Lower cutoff frequency

The lower passband or stopband frequency for the **Bandpass** and **Bandstop** filter configurations. A value of 1 specifies half the sample frequency. Tunable.

Upper cutoff frequency

The upper passband or stopband frequency for the **Bandpass** and **Bandstop** filter configurations. A value of 1 specifies half the sample frequency. Tunable.

Cutoff frequency vector

A vector of ascending frequency points defining the cutoff edges for the **Multiband** filter. A value of 1 specifies half the sample frequency. Tunable.

Gain in the first band

The gain in the first band of the **Multiband** filter: 0 specifies a stopband, 1 specifies a passband. Additional bands alternate between passband and stopband. Tunable.

Frequency vector

A vector of ascending frequency points defining the frequency bands of the **Arbitrary shape** filter. The frequency range is 0 to 1 including the endpoints, where 1 corresponds to half the sample frequency. Tunable.

Gains at these frequencies

A vector containing the desired magnitude response for the **Arbitrary shape** filter at the corresponding points in the **Frequency vector**. Tunable.

Window type

The type of window to apply. See the Window Function block reference. Tunable.

Stopband ripple

The level (dB) of stopband ripple, R_s, for the **Chebyshev** window. Tunable.

Digital FIR Filter Design

Beta

The **Kaiser** window β parameter. Increasing **Beta** widens the mainlobe and decreases the amplitude of the window sidelobes in the window's frequency magnitude response. Tunable.

References

Antoniou, A. *Digital Filters: Analysis, Design, and Applications*. 2nd ed. New York, NY: McGraw-Hill, 1993.

Oppenheim, A. V. and R. W. Schafer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989.

Proakis, J. and D. Manolakis. *Digital Signal Processing*. 3rd ed. Englewood Cliffs, NJ: Prentice-Hall, 1996.

Purpose   Design and implement a raised cosine FIR filter.

Library   dspobslib

Description

The Digital FIR Raised Cosine Filter Design block uses the firrcos function in the Signal Processing Toolbox to design a lowpass, linear-phase, digital FIR filter with a raised cosine transition band. The block applies the filter to a discrete-time input using the Direct-Form II Transpose Filter block.

An *M*-by-*N* sample-based matrix input is treated as *M**N independent channels, and an *M*-by-*N* frame-based matrix input is treated as *N* independent channels. In both cases, the block filters each channel independently over time, and the output has the same size and frame status as the input.

The frequency response of the raised cosine filter is

$$
H(f) = \begin{cases} \dfrac{1}{2f_{n0}} & 0 \le |f| \le (1-R)f_{n0} \\[2em] \left[\dfrac{1 + \cos\dfrac{\pi}{2Rf_{n0}}\left|f\right| - (1-R)f_{n0}}{4f_{n0}}\right] & (1-R)f_{n0} \le |f| \le (1+R)f, \\[2em] 0 & (1+R)f_{n0} \le |f| \le 1 \end{cases}
$$

where $H(f)$ is the magnitude response at frequency f, f_{n0} is the normalized cutoff frequency (-6 dB) specified by the **Upper cutoff**

Digital FIR Raised Cosine Filter Design

frequency parameter, and R is a rolloff factor in the range [0,1] determining the passband-to-stopband transition width.

The **Square-root raised cosine filter** option designs a filter with magnitude response $\sqrt{H(f)}$. This is useful when the filter is part of a pair of matched filters.

When the **Design method** parameter is set to **Rolloff factor**, the secondary **Rolloff factor** parameter is enabled, and R can be directly specified. When **Design method** is set to **Transition bandwidth**, the secondary **Transition bandwidth** parameter is enabled, and the transition region bandwidth, Δf, can be specified in place of R. The transition region is centered on f_{n0} and must be sufficiently narrow to satisfy

$$0 < \left(f_{n0} \pm \frac{\Delta f}{2} \right) < 1$$

The **Upper cutoff frequency** and **Transition bandwidth** parameter values are normalized to half the sample frequency.

The **Window type** parameter allows you to apply a variety of different windows to the raised cosine filter. See the Window Function block reference for a complete description of the available options.

Algorithm The filter output is computed by convolving the input with a truncated, delayed, windowed version of the filter's impulse response. The impulse response for the raised cosine filter is

$$h(kT_s) = \frac{1}{F_s} \text{sinc}(2kT_s f_{n0}) \frac{\cos(2\pi Rk T_s f_{n0})}{1 - (4Rk T_s f_{n0})^2} \qquad -\infty < k < \infty$$

which has limits

$$h(0) = \frac{1}{F_s}$$

and

$$h\left(\pm\frac{1}{4Rf_{n0}}\right) = \frac{R}{2F_s}\sin\left(\frac{\pi}{2R}\right)$$

The impulse response for the square-root raised cosine filter is

$$h(kT_s) = \frac{4R\cos((1+R)2\pi kT_sf_{n0}) + \dfrac{\sin((1-R)2\pi kT_sf_{n0})}{8RkT_sf_{n0}}}{\pi F_s\sqrt{\dfrac{1}{2f_{n0}}}((8RkT_sf_{n0})^2 - 1)} \qquad -\infty < k < \infty$$

which has limits

$$h(0) = (-4R - \pi + \pi R)\frac{\sqrt{2f_{n0}}}{\pi F_s}$$

and

$$\left(\pm\frac{1}{8Rf_{n0}}\right) = -\frac{\sqrt{2f_{n0}}}{2\pi F_s}\left[\pi(1+R)\sin\left(\frac{\pi(1+R)}{4R}\right) - 4R\sin\left(\frac{\pi(R-1)}{4R}\right)\right.$$
$$\left. + \pi(R-1)\cos\left(\frac{\pi(R-1)}{4R}\right)\right]$$

Digital FIR Raised Cosine Filter Design

Dialog Box

Filter order
> The order of the filter. The filter length is one more than this value.

Upper cutoff frequency
> The normalized cutoff frequency, f_{n0}. A value of 1 specifies half the sample frequency. Tunable.

Square-root raised cosine filter
> Selects the square-root filter option, which designs a filter with magnitude response $\sqrt{H(f)}$. Tunable.

Design method
> The method used to design the transition region of the filter, **Rolloff factor** or **Transition bandwidth**. Tunable.

Rolloff factor

The rolloff factor, *R*, enabled when **Rolloff factor** is selected in the **Design method** parameter. Tunable.

Transition bandwidth

The transition bandwidth, Δf, enabled when **Transition bandwidth** is selected in the **Design method** parameter. Tunable.

Window type

The type of window to apply. See the Window Function block reference. Tunable.

Stopband attenuation in dB

The level (dB) of stopband attenuation, R_s, for the Chebyshev window. Tunable.

Beta

The **Kaiser** window β parameter. Increasing β widens the mainlobe and decreases the amplitude of the window sidelobes in the window's frequency magnitude response. Tunable.

Initial conditions

The filter's initial conditions, a scalar, vector, or matrix. See the Direct-Form II Transpose Filter block reference for complete syntax information.

References

Proakis, J. G. *Digital Communications.* Third ed. New York, NY: McGraw-Hill, 1995.
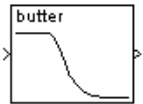
Proakis, J. G. and M. Salehi. *Contemporary Communication Systems Using MATLAB.* Boston, MA: PWS Publishing, 1998.

Digital IIR Filter Design

Purpose   Design and implement an IIR filter.

Library   dspobslib

Description

Note The Digital IIR Filter Design block is still supported but is likely to be obsoleted in a future release. We strongly recommend replacing this block with the Digital Filter block.

The Digital IIR Filter Design block designs a discrete-time (digital) IIR filter in a lowpass, highpass, bandpass, or bandstop configuration, and applies it to the input using the Direct-Form II Transpose Filter block.

An *M*-by-*N* sample-based matrix input is treated as *M*N* independent channels, and an *M*-by-*N* frame-based matrix input is treated as *N* independent channels. In both cases, the block filters each channel independently over time, and the output has the same size and frame status as the input.

The **Design method** parameter allows you to specify Butterworth, Chebyshev type I, Chebyshev type II, and elliptic filter designs. Note that for the bandpass and bandstop configurations, the actual filter length is twice the **Filter order** parameter value.

Filter Design	Description
Butterworth	The magnitude response of a Butterworth filter is maximally flat in the passband and monotonic overall.
Chebyshev type I	The magnitude response of a Chebyshev type I filter is equiripple in the passband and monotonic in the stopband.

Filter Design	Description
Chebyshev type II	The magnitude response of a Chebyshev type II filter is monotonic in the passband and equiripple in the stopband.
Elliptic	The magnitude response of an elliptic filter is equiripple in both the passband and the stopband.

The design and band configuration of the filter are selected from the **Design method** and **Filter type** pop-up menus in the dialog box. For each combination of design method and band configuration, an appropriate set of secondary parameters is displayed.

The table below lists the available parameters for each design/band combination. For lowpass and highpass band configurations, these parameters include the passband edge frequency f_{np}, the stopband edge frequency f_{ns}, the passband ripple R_p, and the stopband attenuation R_s. For bandpass and bandstop configurations, the parameters include the lower and upper passband edge frequencies, f_{np1} and f_{np2}, the lower and upper stopband edge frequencies, f_{ns1} and f_{ns2}, the passband ripple R_p, and the stopband attenuation R_s. Frequency values are normalized to half the sample frequency, and ripple and attenuation values are in dB.

	Lowpass	**Highpass**	**Bandpass**	**Bandstop**
Butterworth	Order, f_{np}	Order, f_{np}	Order, f_{np1}, f_{np2}	Order, f_{np1}, f_{np2}
Chebyshev Type I	Order, f_{np}, R_p	Order, f_{np}, R_p	Order, f_{np1}, f_{np2}, R_p	Order, f_{np1}, f_{np2}, R_p
Chebyshev Type II	Order, f_{ns}, R_s	Order, f_{ns}, R_s	Order, f_{ns1}, f_{ns2}, R_s	Order, f_{ns1}, f_{ns2}, R_s
Elliptic	Order, f_{np}, R_p, R_s	Order, f_{np}, R_p, R_s	Order, f_{np1}, f_{np2}, R_p, R_s	Order, f_{np1}, f_{np2}, R_p, R_s

The digital filters are designed using the Signal Processing Toolbox's filter design commands `butter`, `cheby1`, `cheby2`, and `ellip`.

Digital IIR Filter Design

**Dialog
Box**
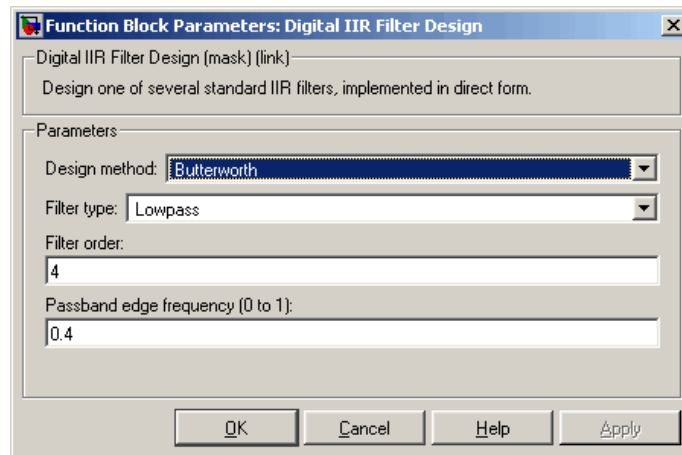
The parameters displayed in the dialog box vary for different
design/band combinations. Only some of the parameters listed below
are visible in the dialog box at any one time.

Design method
> The filter design method: `Butterworth`, `Chebyshev type I`,
> `Chebyshev type II`, or `Elliptic`. Tunable.

Filter type
> The type of filter to design: `Lowpass`, `Highpass`, `Bandpass`, or
> `Bandstop`. Tunable.

Filter order
> The order of the filter for lowpass and highpass configurations.
> For bandpass and bandstop configurations, the length of the final
> filter is twice this value.

Passband edge frequency
> The normalized passband edge frequency for the highpass and
> lowpass configurations of the Butterworth, Chebyshev type I, and
> elliptic designs. Tunable.

Lower passband edge frequency

The normalized lower passband frequency for the bandpass and bandstop configurations of the Butterworth, Chebyshev type I, and elliptic designs. Tunable.

Upper passband edge frequency

The normalized upper passband frequency for the bandpass and bandstop configurations of the Butterworth, Chebyshev type I, or elliptic designs. Tunable.

Stopband edge frequency

The normalized stopband edge frequency for the highpass and lowpass band configurations of the Chebyshev type II design. Tunable.

Lower stopband edge frequency

The normalized lower stopband frequency for the bandpass and bandstop configurations of the Chebyshev type II design. Tunable.

Upper stopband edge frequency

The normalized upper stopband frequency for the bandpass and bandstop filter configurations of the Chebyshev type II design. Tunable.

Passband ripple in dB

The passband ripple, in dB, for the Chebyshev type I and elliptic designs. Tunable.

Stopband attenuation in dB

The stopband attenuation, in dB, for the Chebyshev type II and elliptic designs. Tunable.

References   Antoniou, A. *Digital Filters: Analysis, Design, and Applications*. 2nd ed. New York, NY: McGraw-Hill, 1993.

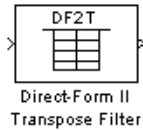Oppenheim, A. V. and R. W. Schafer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989.

Proakis, J. and D. Manolakis. *Digital Signal Processing*. 3rd ed. Englewood Cliffs, NJ: Prentice-Hall, 1996.

Direct-Form II Transpose Filter

Purpose   Apply an IIR filter to the input.

Library   dspobslib

Description

Direct-Form II
Transpose Filter

Note The Direct-Form II Transpose Filter block is still supported but is likely to be obsoleted in a future release. We strongly recommend replacing this block with the Digital Filter block.

The Direct-Form II Transpose Filter block applies a transposed direct-form II IIR filter to the input.
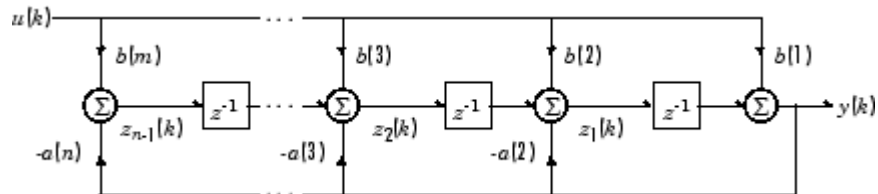
This is a canonical form that has the minimum number of delay elements. The filter order is max(m,n)-1.

An *M*-by-*N* sample-based matrix input is treated as *M*N* independent channels, and an *M*-by-*N* frame-based matrix input is treated as *N* independent channels. In both cases, the block filters each channel independently over time, and the output has the same size and frame status as the input.

The filter is specified in the parameter dialog box by its transfer function,

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_1 + b_2 z^{-1} + \ldots + b_{m+1} z^{-(m-1)}}{a_1 + a_2 z^{-1} + \ldots + a_{n+1} z^{-(n-1)}}$$

where the **Numerator** parameter specifies the vector of numerator coefficients,

```
[b(1) b(2) ... b(m)]
```

and the **Denominator** parameter specifies the vector of denominator coefficients,

```
[a(1) a(2) ... a(n)]
```

The filter coefficients are normalized by a_1.

Initial Conditions

In its default form, the filter initializes the internal filter states to zero, which is equivalent to assuming past inputs and outputs are zero. The block also accepts optional nonzero initial conditions for the filter delays. Note that the number of filter states (delay elements) per input channel is

```
max(m,n)-1
```

The **Initial conditions** parameter may take one of four forms:

- Empty matrix

  The empty matrix, [ ], causes a zero (0) initial condition to be applied to all delay elements in each filter channel.

- Scalar

  The scalar value is copied to all delay elements in each filter channel. Note that a value of zero is equivalent to setting the **Initial conditions** parameter to the empty matrix, [ ].
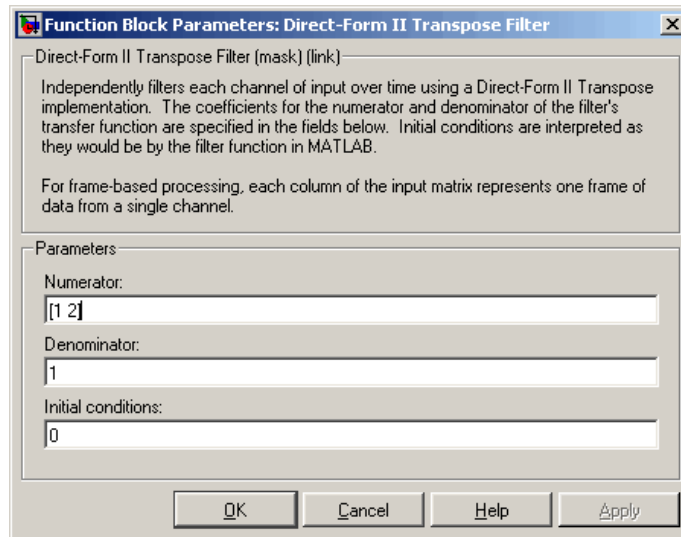
- Vector

  The vector has a length equal to the number of delay elements in each filter channel, `max(m,n)-1`, and specifies a unique initial condition for each delay element in the filter channel. This vector of initial conditions is applied to each filter channel.

Direct-Form II Transpose Filter

- Matrix

  The matrix specifies a unique initial condition for each delay element, and can specify different initial conditions for each filter channel. The matrix must have the same number of rows as the number of delay elements in the filter, max(m,n)-1, and must have one column per filter channel.

Dialog Box

Numerator

The filter numerator vector. Tunable; the numerator coefficients can be adjusted while the simulation runs, but the vector length (i.e., the filter order) must remain the same.

Denominator

The filter denominator vector. Tunable; the denominator coefficients can be adjusted while the simulation runs, but the vector length (i.e., the filter order) must remain the same.

Initial conditions

The filter's initial conditions, a scalar, vector, or matrix.

References   Oppenheim, A. V. and R. W. Schafer. *Discrete-Time Signal Processing*.
Englewood Cliffs, NJ: Prentice Hall, 1989.

Proakis, J. and D. Manolakis. *Digital Signal Processing.* 3rd ed.
Englewood Cliffs, NJ: Prentice-Hall, 1996.

Discrete Impulse

Purpose   Generate discrete impulse

Library   Signal Processing Sources

Description The Discrete Impulse block generates an impulse (the value 1) at output
sample D+1, where D is specified by the **Delay** parameter (D ≥0). All
output samples preceding and following sample D+1 are zero.

When D is a length-*N* vector, the block generates an *M*-by-*N* matrix
output representing *N* distinct channels, where frame size *M* is specified
by the **Samples per frame** parameter. The impulse for the ith channel
appears at sample D(i)+1. For *M*=1, the output is sample based;
otherwise, the output is frame based.

The **Sample time** parameter value, T_s, specifies the output signal
sample period. The resulting frame period is $M*T_s$.

Examples   Construct the model below.
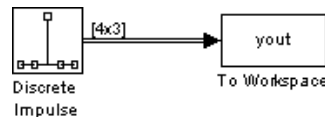
Configure the Discrete Impulse block to generate a frame-based
three-channel output of type double, with impulses at samples 1, 4, and
6 of channels 1, 2, and 3, respectively. Use a sample period of 0.25 and a
frame size of 4. The corresponding settings should be as follows:

- **Delay** = [0 3 5]
- **Sample time** = 0.25
- **Samples per frame** = 4
- **Output data type** = double

Run the model and look at the output, yout. The first few samples of each channel are shown below.

```
yout(1:10,:)
ans =
     1     0     0
     0     0     0
     0     0     0
     0     1     0
     0     0     0
     0     0     1
     0     0     0
     0     0     0
     0     0     0
     0     0     0
```

The block generates an impulse at sample 1 of channel 1 (first column), at sample 4 of channel 2 (second column), and at sample 6 of channel 3 (third column).

Dialog Box

The **Main** pane of the Discrete Impulse block dialog appears as follows:
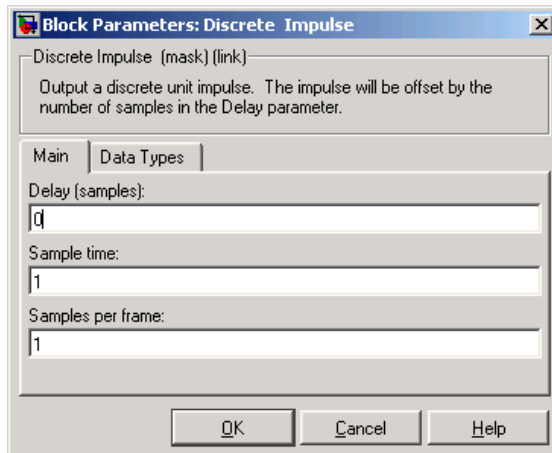
Discrete Impulse

Delay
> The number of zero-valued output samples, D, preceding the impulse. A length-*N* vector specifies an *N*-channel output. This parameter is not tunable.
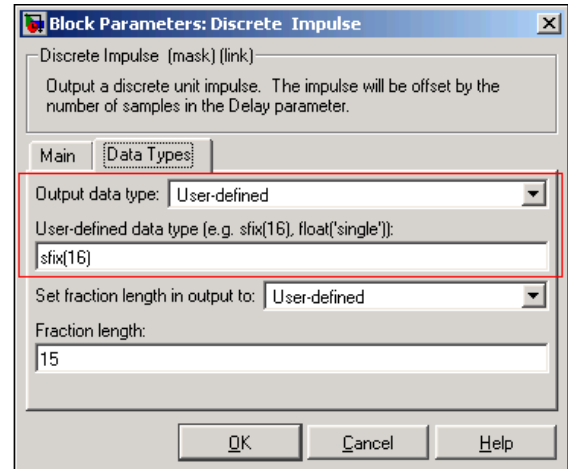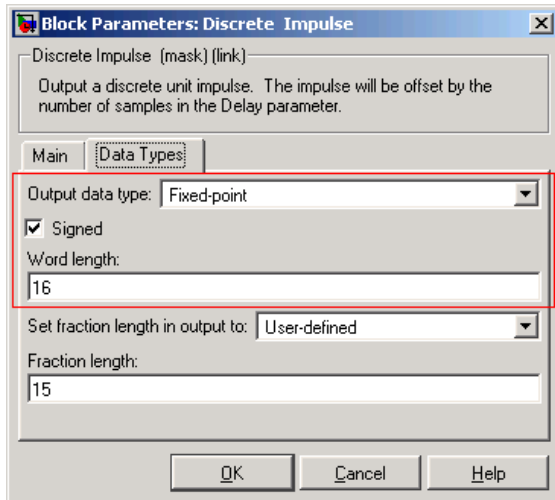
Sample time
> The sample period, T_s, of the output signal. The output frame period is $M*T_s$. This parameter is not tunable.

Samples per frame
> The number of samples, *M*, in each output frame. This parameter is not tunable.

The **Data Types** pane of the Discrete Impulse block dialog appears as follows:

Output data type
> Specify the output data type in one of the following ways:
>
> • Choose one of the built-in data types from the list.

- Choose `Fixed-point` to specify the output data type and scaling in the **Signed**, **Word length**, **Set fraction length in output to,** and **Fraction length** parameters.

- Choose `User-defined` to specify the output data type and scaling in the **User-defined data type**, **Set fraction length in output to,** and **Fraction length** parameters.

- Choose `Inherit via back propagation` to set the output data type and scaling to match the next block downstream.

Signed

    Select to output a signed fixed-point signal. Otherwise, the signal will be unsigned. This parameter is only visible when you select `Fixed-point` for the **Output data type** parameter.

Word length

    Specify the word length, in bits, of the fixed-point output data type. This parameter is only visible when you select `Fixed-point` for the **Output data type** parameter.

User-defined data type

    Specify any built-in or fixed-point data type. You can specify fixed-point data types using the `sfix`, `ufix`, `sint`, `uint`, `sfrac`, and `ufrac` functions from Simulink Fixed Point. This parameter is only visible when you select `User-defined` for the **Output data type** parameter.

Set fraction length in output to

    Specify the scaling of the fixed-point output by either of the following two methods:

- Choose `Best precision` to have the output scaling automatically set such that the output signal has the best possible precision.

- Choose `User-defined` to specify the output scaling in the **Fraction length** parameter.

    This parameter is only visible when you select `Fixed-point` for the **Output data type** parameter, or when you select

Discrete Impulse

User-defined and the specified output data type is a fixed-point data type.

Fraction length
For fixed-point output data types, specify the number of fractional bits, or bits to the right of the binary point. This parameter is only visible when you select Fixed-point or User-defined for the **Output data type** parameter and User-defined for the **Set fraction length in output to** parameter.

Supported Data Types
- Double-precision floating point
- Single-precision floating point
- Fixed point
- Boolean
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Data Type Conversion	Simulink
DSP Constant	Signal Processing Blockset
Multiphase Clock	Signal Processing Blockset
N-Sample Enable	Signal Processing Blockset
Signal From Workspace	Signal Processing Blockset
impz	Signal Processing Toolbox

Purpose   Resample input at lower rate by deleting samples

Library   Signal Operations

Description   The Downsample block resamples each channel of the M_i-by-N input
at a rate K times lower than the input sample rate by discarding
K-1 consecutive samples following each sample passed through to
the output. The integer K is specified by the **Downsample factor**
parameter.

The **Sample offset** parameter delays the output samples by an integer
number of sample periods, D, where $0 \leq D < (K-1)$, so that any of the
K possible output phases can be selected. For example, when you
downsample the sequence 1, 2, 3, ... by a factor of 4, you can select from
the following four phases.

Input Sequence	Sample Offset, D	Output Sequence (K=4)
1,2,3,...	0	0,1,5,9,13,17,21,25,...
1,2,3,...	1	0,2,6,10,14,18,22,26,...
1,2,3,...	2	0,3,7,11,15,19,23,27,...
1,2,3,...	3	0,4,8,12,16,20,24,28,...

The initial zero in each output sequence above is a result of the default
zero **Initial condition** parameter setting for this example. See
"Latency" on page 10-328 for more on the **Initial condition** parameter.

This block supports triggered subsystems if, for **Sample-based mode**,
you select Force single-rate and, for **Frame-based mode**, you select
Maintain input frame rate.

Sample-Based Operation

When the input is sample based, the block treats each of the $M*N$
matrix elements as an independent channel, and downsamples each
channel over time. The input and output sizes are identical.

Downsample

The **Sample-based mode** parameter determines how the block represents the new rate at the output. There are two available options:

- `Allow multirate`

  When you select `Allow multirate`, the sample period of the sample-based output is K times longer than the input sample period ($T_{so} = KT_{si}$). The block is therefore multirate.

- `Force single rate`

  When you select `Force single rate`, the block forces the output sample rate to match the input sample rate ($T_{so} = T_{si}$) by repeating every Kth input sample K times at the output. The block is therefore single-rate. (The block's operation when you select `Enforce single rate` is similar to the operation of a Sample and Hold block with a repeating trigger event of period KT_{si}.)

The setting of the **Frame-based mode** pop-up menu does not affect sample-based inputs.

Frame-Based Inputs

When the input is frame based, the block treats each of the N input columns as a frame containing M_i sequential time samples from an independent channel. The block downsamples each channel independently by discarding K-1 rows of the input matrix following each row that it passes through to the output.

The **Frame-based mode** parameter determines how the block adjusts the rate at the output to accommodate the reduced number of samples. There are two available options:

- `Maintain input frame size`

  The block generates the output at the slower (downsampled) rate by using a proportionally longer frame *period* at the output port than at the input port. For downsampling by a factor of K, the output frame period is K times longer than the input frame period ($T_{fo} = KT_{fi}$), but the input and output frame sizes are equal.

The model below shows a single-channel input with a frame period of 1 second being downsampled by a factor of 4 to a frame period of 4 seconds. The input and output frame sizes are identical.
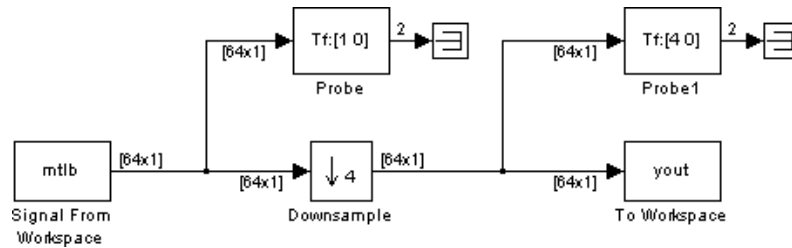
- Maintain input frame rate

  The block generates the output at the slower (downsampled) rate by using a proportionally smaller frame *size* than the input. For downsampling by a factor of *K*, the output frame size is *K* times smaller than the input frame size ($M_o = M_i/K$), but the input and output frame rates are equal.

  The model below shows a single-channel input of frame size 64 being downsampled by a factor of 4 to a frame size of 16. The input and output frame rates are identical.
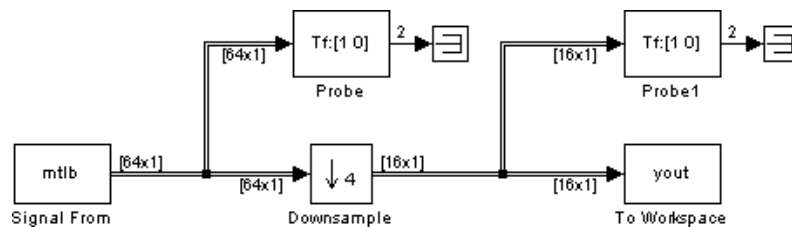
The setting of the **Sample-based mode** pop-up menu does not affect frame-based inputs.

Downsample

Latency

Zero Latency

The Downsample block has *zero tasking latency* for the special combinations of input signal sampling and parameter settings shown in the table below. In all of these cases the block has single-rate operation.

Input Sampling	Parameter Settings
Sample-based	**Downsample factor** parameter, *K*, is 1, *or* **Enforce single rate** is selected (with D=0)
Frame-based	**Downsample factor** parameter, *K*, is 1, *or* **Maintain input frame rate** is selected

Zero tasking latency means that the block propagates input sample $D+1$ (received at t=0) as the first output sample, followed by input sample $D+1+K$, input sample $D+1+2K$, and so on. The **Initial condition** parameter value is not used.

Nonzero Latency

The Downsample block is multirate for most settings other than those in the above table. The amount of latency for multirate operation depends on input signal sampling and the Simulink tasking mode, as shown in the table below.

Multirate...	Sample-Based Latency	Frame-Based Latency
Single-tasking	None, for D=0 One sample, for D>0	One frame (M_i samples)
Multitasking	One sample	One frame (M_i samples)

The only case of nonzero single-rate latency occurs in sample-based mode, when you select Force single rate with $D > 0$. The latency in this case is one sample.

In all cases of *one-sample latency*, the initial condition for each channel appears as the first output sample. Input sample $D+1$ appears as the second output sample for each channel, followed by input sample $D+1+K$, input sample $D+1+2K$, and so on. The **Initial condition** parameter can be an M_i-by-N matrix containing one value for each channel, or a scalar to be applied to all signal channels.

In all cases of *one-frame latency*, the M_i rows of the initial condition matrix appear in sequence as the first M_i output rows. Input sample $D+1$ (i.e, row $D+1$ of the input matrix) appears in the output as sample M_i+1, followed by input sample $D+1+K$, input sample $D+1+2K$, and so on. The **Initial condition** value can be an M_i-by-N matrix, or a scalar to be repeated across all elements of the M_i-by-N matrix. See the following example for an illustration of this case.

Note For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" on page 2-56 and "Models with Multiple Sample Rates" in the Real-Time Workshop User's Guide documentation.

Examples   Construct the frame-based model shown below.
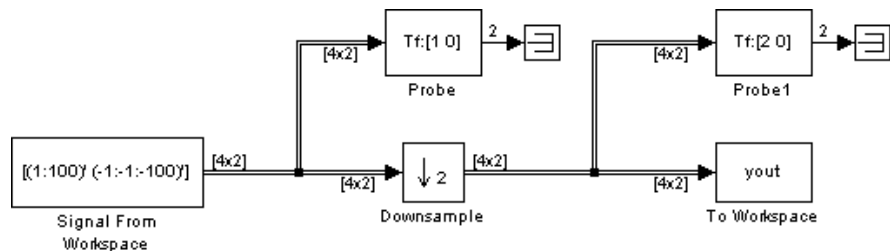
Adjust the block parameters as follows:

- Configure the Signal From Workspace block to generate a two-channel signal with frame size of 4 and sample period of 0.25 second. This represents an output frame period of 1 second (0.25*4).

Downsample

The first channel should contain the positive ramp signal 1, 2, ...,
100, and the second channel should contain the negative ramp signal
-1, -2, ..., -100. The settings are

- **Signal** = `[(1:100)' (-1:-1:-100)']`

- **Sample time** = `0.25`

- **Samples per frame** = `4`

• Configure the Downsample block to downsample the two-channel
  input by decreasing the output frame rate by a factor of 2 relative to
  the input frame rate. Set a sample offset of 1, and a 4-by-2 initial
  condition matrix of

$$\begin{bmatrix} 11 & -11 \\ 12 & -12 \\ 13 & -13 \\ 14 & -14 \end{bmatrix}$$

- **Downsample factor** = `2`

- **Sample offset** = `1`

- **Initial condition** = `[11 -11;12 -12;13 -13;14 -14]`

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

• Configure the Probe blocks by clearing the **Probe width** and **Probe
  complex signal** check boxes (if desired).

This model is multirate because there are at least two distinct frame
rates, as shown by the two Probe blocks. To run this model in the
Simulink multitasking mode, open the Configuration Parameters dialog
box. From the list on the left side of the dialog box, click **Solver**. From
the **Type** list, select `Fixed-step`, and from the **Solver** list, select
`discrete (no continuous states)`. From the **Tasking mode for
periodic sample times** list, select `MultiTasking`. Additionally, set
the **Stop time** parameter to `30`.

Run the model and look at the output, yout. The first few samples of each channel are shown below.

```
yout =
    11    -11
    12    -12
    13    -13
    14    -14
     2     -2
     4     -4
     6     -6
     8     -8
    10    -10
    12    -12
    14    -14
```

Since we ran this frame based multirate model in multitasking mode, the first row of the initial condition matrix appears as the first output sample, followed by the other three initial condition rows. The second row of the first input matrix (that is, row D+1, where D is the **Sample offset**) appears in the output as sample 5 (that is sample M_i+1, where M_i is the input frame size).

Downsample

Dialog Box
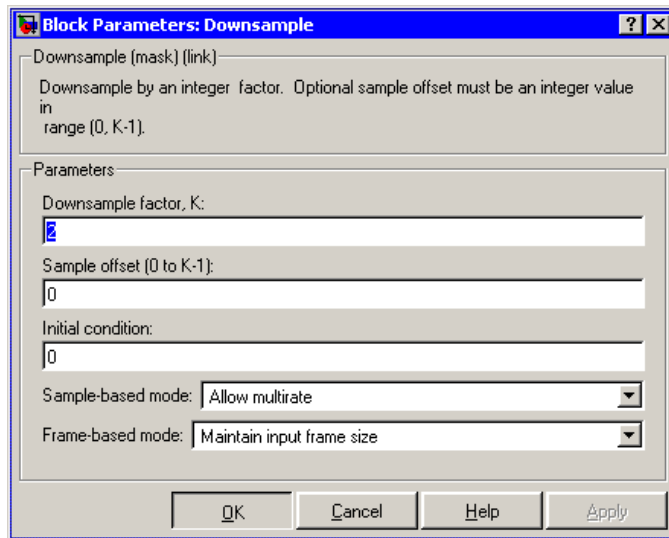
Downsample factor

The integer factor, *K*, by which to decrease the input sample rate.

Sample offset

The sample offset, *D*, which must be an integer in the range [0, *K*-1].

Initial condition

The value with which the block is initialized for cases of nonzero latency; a scalar or matrix.

Sample-based mode

The method by which to implement downsampling for sample-based inputs: Allow multirate (that is, decrease the output sample rate), or Force single-rate (that is, force the output sample rate to match the input sample rate by repeating every *K*th input sample *K* times at the output).

Frame-based mode

The method by which to implement downsampling for frame-based inputs: Maintain input frame size (that is, decrease the frame

rate), or `Maintain input frame rate` (that is, decrease the frame size).

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Custom data types
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Custom data types
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

Downsample

See Also

FIR Decimation	Signal Processing Blockset
FIR Rate Conversion	Signal Processing Blockset
Repeat	Signal Processing Blockset
Sample and Hold	Signal Processing Blockset
Upsample	Signal Processing Blockset

Purpose   Generate discrete- or continuous-time constant signal

Library   Signal Processing Sources

Description   The DSP Constant block generates a signal whose value remains
constant throughout the simulation. The **Constant value** parameter
specifies the constant to output, and can be any valid MATLAB
expression that evaluates to a scalar, vector, or matrix.

When **Sample mode** is set to Continuous, the output is a
continuous-time signal. When **Sample mode** is set to Discrete, the
Sample time parameter is visible, and the signal has the discrete
output period specified by the **Sample time** parameter.

You can set the output signal to Frame-based, Sample-based, or
Sample-based (interpret vectors as 1-D) with the **Output**
parameter.

Dialog   The **Main** pane of the DSP Constant block dialog box appears as follows:
Box
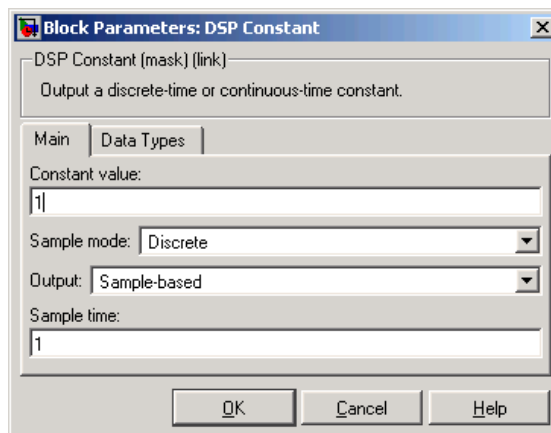
Opening this dialog box causes a running simulation to pause.
See "Changing Source Block Parameters" in the online Simulink
documentation for details.

Constant value
> Specify the constant to generate. This parameter is tunable; values entered here can be tuned, but their dimensions must remain fixed.
>
> When you specify any data type information in this field, it is overridden by the value of the **Output data type** parameter in the **Data Types** pane, unless you select `Inherit from 'Constant value'`.

Sample mode
> Specify the sample mode of the output, `Discrete` for a discrete-time signal or `Continuous` for a continuous-time signal.

Output
> Specify whether the output is `Sample-based (interpret vectors as 1-D)`, `Sample-based`, or `Frame-based`. When you select `Sample-based` and the output is a vector, its dimension is constrained to match the **Constant value** dimension (row or column). When you select `Sample-based (interpret vectors as 1-D)`, however, the output has no specified dimensionality.

Sample time
> Specify the discrete sample period for sample-based outputs. When you select `Frame-based` for the **Output** parameter, this parameter is named **Frame period**, and is the discrete frame period for the frame-based output. This parameter is only visible when you select `Discrete` for the **Sample mode** parameter.

The **Data Types** pane of the DSP Constant block dialog box appears as follows:
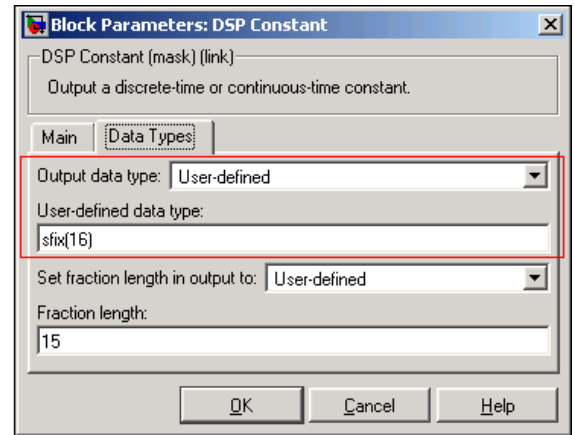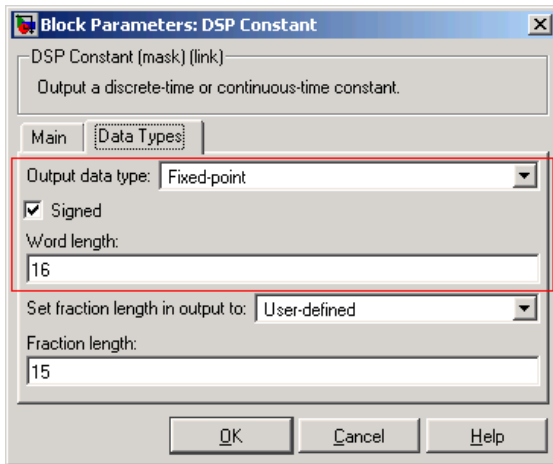
Output data type

Specify the output data type in one of the following ways:

- Choose one of the built-in data types from the list.

- Choose Fixed-point to specify the output data type and scaling in the **Signed**, **Word length**, **Set fraction length in output to,** and **Fraction length** parameters.

- Choose User-defined to specify the output data type and scaling in the **User-defined data type**, **Set fraction length in output to,** and **Fraction length** parameters.

- Choose Inherit from 'Constant value' to set the output data type and scaling to match the values of the **Constant value** parameter in the **Main** pane.

- Choose Inherit via back propagation to set the output data type and scaling to match the following block.

The value of this parameter overrides any data type information specified in the **Constant value** parameter in the **Main** pane, except when you select `Inherit from 'Constant value'`.

Signed

Select to output a signed fixed-point signal. Otherwise, the signal will be unsigned. This parameter is only visible when you select `Fixed-point` for the **Output data type** parameter.

Word length

Specify the word length, in bits, of the fixed-point output data type. This parameter is only visible when you select `Fixed-point` for the **Output data type** parameter.

User-defined data type

Specify any built-in or fixed-point data type. You can specify fixed-point data types using the `sfix`, `ufix`, `sint`, `uint`, `sfrac`, and `ufrac` functions from Simulink Fixed Point. This parameter is only visible when you select `User-defined` for the **Output data type** parameter.

Set fraction length in output to

Specify the scaling of the fixed-point output by either of the following two methods:

- Choose `Best precision` to have the output scaling automatically set such that the output signal has the best possible precision.

- Choose `User-defined` to specify the output scaling in the **Fraction length** parameter.

This parameter is only visible when you select `Fixed-point` for the **Output data type** parameter, or when you select `User-defined` and the specified output data type is a fixed-point data type.

Fraction length

For fixed-point output data types, specify the number of fractional bits, or bits to the right of the binary point. This parameter is

only visible when you select `Fixed-point` or `User-defined` for the **Output data type** parameter and `User-defined` for the **Set fraction length in output to** parameter.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point
- Custom data types
- Boolean
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Constant	Simulink
Signal From Workspace	Signal Processing Blockset

DSP Fixed-Point Attributes

Purpose   Set fixed-point attributes of Signal Processing Blockset blocks on the system or subsystem level

Library   dspobslib

Description

DSP Fixed-Point
Attributes

Note The DSP Fixed-Point Attributes (DFPA) block is still supported but is likely to be obsoleted in a future release.

The DSP Fixed-Point Attributes (DFPA) block enables you to set fixed-point attributes for Signal Processing Blockset blocks in your model on the system or subsystem level. This allows you to set fixed-point parameters for groups of blocks in one place, rather than on a block-by-block basis. The parameters listed below appear on various fixed-point Signal Processing Blockset block masks, and can be controlled by DFPA blocks.

On nonsource blocks:

- **Output word length**
- **Output fraction length**
- **Accumulator word length**
- **Accumulator fraction length**
- **Product output word length**
- **Product output fraction length**
- **State memory word length**
- **State memory fraction length**
- **Round integer calculations toward**
- **Saturate on integer overflow**

On source blocks:

- **Word length**
- **Set fraction length in output to**
- **Fraction length**

The blocks that have parameters that may be controlled by DFPA blocks are listed below:

- Autocorrelation
- Constant Diagonal Matrix
- Convolution
- Correlation
- Digital Filter
- Discrete Impulse
- DSP Constant
- DSP Gain
- DSP Product
- DSP Sum

- FFT
- FIR Decimation
- FIR Interpolation
- Identity Matrix
- IFFT
- Matrix Product
- Matrix Scaling
- Matrix Sum
- Sine Wave
- Window Function

Each of these blocks has an **Allow overrides from DSP Fixed-Point Attributes blocks** check box that is selected by default. That means that any such blocks in models you build can automatically be configured from the top level of your model, without having to configure each block mask. If you do not want the fixed-point parameters of a particular block in your model to be controlled by DFPA blocks, clear the **Allow overrides from DSP Fixed-Point Attributes blocks** check box in that block's mask.

Place a DFPA block in any subsystem in your model that contains blocks with any of the fixed-point parameters listed above, if you want to control the blocks at the subsystem level. You can have a DFPA block

DSP Fixed-Point Attributes

in one, some, or all of the subsystems in your model. DFPA blocks in lower subsystems overrule the settings of DFPA blocks at higher levels.

Dialog Box

When you double-click on a DFPA block in any model you have open, the DSP Fixed-Point Attributes GUI appears. This one instance of the GUI enables you to see the information for all DFPA blocks in open models. You can see the hierarchy of DFPA blocks in the left pane of the GUI, and settings for a particular DFPA block in the right pane of the GUI.
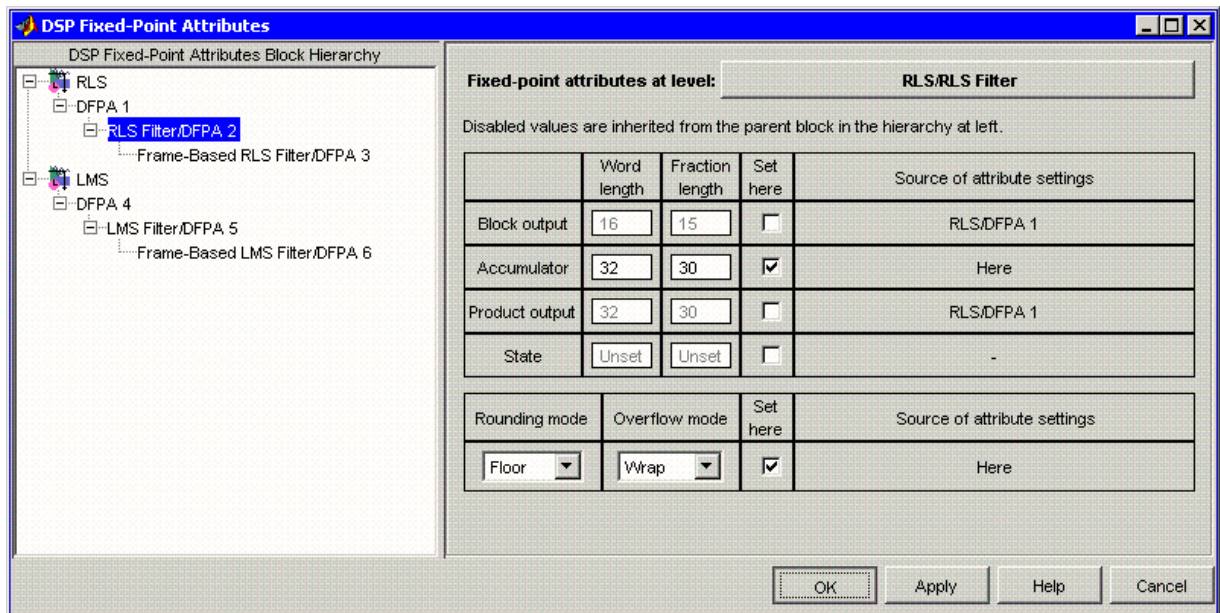
Left Pane

The left pane of the DSP Fixed-Point Attributes GUI displays the DSP Fixed-Point Attributes Block Hierarchy. This navigation tree displays the relative hierarchy of all DFPA blocks in models that are currently open.

Note The left pane displays the hierarchy of *DFPA blocks* in all models that are currently open. It does not display the hierarchy of *subsystems* in your models.

The top-level nodes in the hierarchy, designated by the ![icon] icon, represent each model that you have open. The branches under each top-level model node show the DFPA blocks in that model. Settings in DFPA blocks that are at a lower level in the hierarchy have precedence over higher-level DFPA blocks for the subsystems that they control. Therefore, a DFPA block controls fixed-point settings for blocks that are in the same subsystem or a lower subsystem, unless

- A lower-level DFPA block overrides the settings.

- A particular fixed-point block does not have its **Allow overrides from DSP Attributes blocks** check box selected.

You can click on any branch in the hierarchy to select it. The information for the DFPA block selected in the left pane is displayed in the right pane.

Right Pane

The following buttons, rows, and columns allow you to specify the settings for the currently selected DFPA block:

Fixed-point attributes at level:
> This button displays the path to the subsystem that contains the DFPA block currently selected in the DFPA Block Hierarchy. Click this button to bring the subsystem to the front of your screen.

Block output
> This row allows you to set fixed-point block output attributes.
>
> This row can override the **Output word length** and **Output fraction length** parameters on nonsource block masks. However,

note that the settings in this row are ignored by the DSP Gain, DSP Product, and DSP Sum blocks.

This row can override the **Word length** and **Fraction length** parameters on source masks, if the **Output data type** parameter of the source block is set to Fixed-point.

The block output word length and fraction length may only be set in this row if the corresponding **Set here** check box is selected.

Accumulator

This row allows you to set fixed-point accumulator attributes. This row can override the **Accumulator word length** and **Accumulator fraction length** parameters on block masks. This row also overrides the **Output word length** and **Output fraction length** parameters for DSP Sum blocks.

The accumulator word length and fraction length may only be set in this row if the corresponding **Set here** check box is selected.

Product Output

This row allows you to set fixed-point product output attributes. This row can override the **Product output word length** and **Product output fraction length** parameters on block masks. This row also overrides the **Output word length** and **Output fraction length** parameters for DSP Gain and DSP Product blocks.

The product output word length and fraction length may only be set in this row if the corresponding **Set here** check box is selected.

State

This row allows you to set fixed-point state memory attributes. This row can override the **State memory word length** and **State memory output fraction length** parameters on block masks.

The state word length and fraction length may only be set in this row if the corresponding **Set here** check box is selected.

Rounding Mode

This column allows you to set the fixed-point rounding mode to `Floor` or `Nearest`. This column can override the **Round integer calculations toward** parameter on block masks.

The rounding mode may only be set in this column if the corresponding **Set here** check box is selected.

Overflow Mode

This column allows you to set the fixed-point overflow mode to `Wrap` or `Saturate`. This column can override the **Saturate on integer overflow** parameter on block masks.

The overflow mode may only be set in this column if the corresponding **Set here** check box is selected.

Word length

This column allows you to set the word length, in bits, for the Block output, Accumulator, Product output, and State attributes. Each word length must be an integer number of bits between 2 and 128.

Fraction length

This column allows you to set the fraction length, in bits, for the Block output, Accumulator, Product output, and State attributes. Each fraction length must be an integer number of bits.

Set here

The check boxes in this column allow you to specify whether each attribute is set by the currently selected DFPA block. If a **Set here** check box is not selected, the corresponding attribute is either set by a higher DFPA block, or is not set at all.

Source of attribute settings

This column indicates the level at which each attribute is set. If the **Set here** check box is selected for a certain row, the **Source of attribute settings** cell in that row is set to `Here`. If the **Set here** check box is not selected for a certain row, the **Source of attribute settings** cell for that row gives the path to the DFPA

block controlling those attributes for the current subsystem. If the parameters in a row are not set in any DFPA block, the **Source of attribute settings** cell for that row has a dash-.
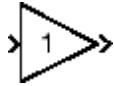
Supported Data Types

This block sets attributes for Signal Processing Blockset blocks that perform fixed-point calculations. This block has no input or output ports.

Purpose   Multiply the input by a constant

Library   dspobslib

Description

> **Note** The DSP Gain block is still supported but is likely to be obsoleted in a future release. We strongly recommend replacing this block with the Simulink Gain block.

The DSP Gain block is a masked version of the Simulink Gain block. It multiplies the input by the gain, element-wise.

The input and the gain can each be a scalar, vector, or matrix. Either the gain must be a scalar, or it must have the same dimensions as the input:

- If the gain is a scalar, then the gain is multiplied to each element of the input.

- If the gain is a vector or a matrix, each element of the gain is multiplied to the corresponding element of the input.

The DSP Gain block accepts real and complex floating-point and fixed-point inputs.

**Dialog
Box**
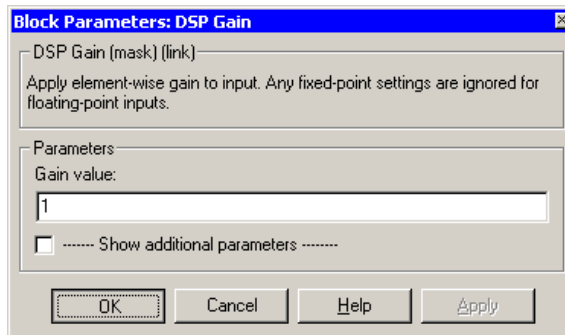
Gain value

Specify the value by which to multiply the input. The gain may be scalar, vector, or a matrix. The gain may not be Boolean.

Show additional parameters

If selected, additional parameters specific to implementation of the block become visible as shown.

Allow overrides from DSP Fixed-Point Attributes blocks

If you select this parameter, fixed-point data types for this block may be set by DSP Fixed-Point Attributes (DFPA) blocks in your

model. If this parameter is unselected, the data types are always set by the parameters in the block mask.

Note that the data type of the gain is always specified by the value in the DSP Gain block mask, and not by a DFPA block.

When a DSP Gain block is overridden by a DFPA block, the output data type of the DSP Gain block is controlled by the **Product output** attribute. The **Output** attribute of the DFPA block is ignored for DSP Gain blocks.

Fixed-point gain attributes
Choose how you will specify the word length and fraction length of the gain. If you select Same as input, these characteristics will match those of the input to the block. If you select User-defined, the **Gain word length** and **Gain fraction length** parameters become visible.

The Gain does not obey the **Round integer calculations toward** and **Saturate on integer overflow** parameters; it is always saturated and rounded to Nearest.

Gain word length
Specify the word length, in bits, of the gain. This parameter is only visible if User-defined is specified for the **Fixed-point gain attributes** parameter.

Gain fraction length
Specify the fraction length, in bits, of the gain. This parameter is only visible if User-defined is specified for the **Fixed-point gain attributes** parameter.

Fixed-point output attributes
Choose how you will specify the output word length and fraction length. If you select Same as input, these characteristics will match those of the input to the block. If you select User-defined, the **Output word length** and **Output fraction length** parameters become visible.

Output word length

> Specify the word length, in bits, of the output. This parameter is only visible if `User-defined` is specified for the **Fixed-point output attributes** parameter.

Output fraction length

> Specify the fraction length, in bits, of the output. This parameter is only visible if `User-defined` is specified for the **Fixed-point output attributes** parameter.

Round integer calculations toward

> Select the rounding mode for fixed-point operations. The gain does not obey this parameter; it always rounds to `Nearest`.

Saturate on integer overflow

> If selected, overflows saturate. The gain does not obey this parameter; it is always saturated.

Supported Data Types

- Double-precision floating point

- Single-precision floating point

- Fixed point

- 8-, 16-, and 32-bit signed integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

DSP Product	Signal Processing Blockset
DSP Sum	Signal Processing Blockset
Gain	Simulink

Purpose   Perform element-wise multiplication of two inputs

Library   dspobslib

Description
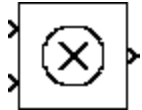
> **Note** The DSP Product block is still supported but is likely to be
> obsoleted in a future release. We strongly recommend replacing this
> block with the Simulink Product block.

The DSP Product block is a masked version of the Simulink Product
block. The DSP Product block performs element-wise multiplication of
two inputs.

The inputs to the DSP Product block can be scalar, vector, or matrix.
Either both inputs must have the same dimensions, or at least one of
the inputs must be a scalar:

- If one input is a scalar, it is multiplied to each element of the other
  input.

- If both inputs are vectors or a matrices, their corresponding elements
  are multiplied together.

The DSP Product block accepts real and complex floating-point and
fixed-point inputs.
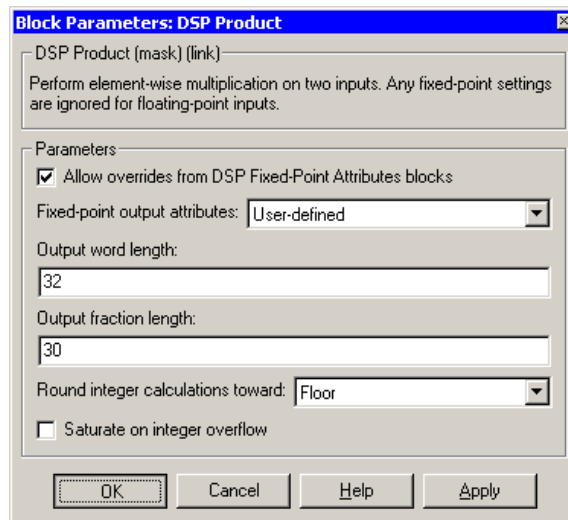
DSP Product

Dialog Box

Allow overrides from DSP Fixed-Point Attributes blocks

If you select this parameter, fixed-point data types for this block may be set by DSP Fixed-Point Attributes (DFPA) blocks in your model. If this parameter is unselected, the data types are always set by the parameters in the block mask.

When a DSP Product block is overridden by a DFPA block, the output data type of the DSP Product block is controlled by the **Product output** attribute. The **Output** attribute of the DFPA block is ignored for DSP Product blocks.

Fixed-point output attributes

Choose how you will specify the output word length and fraction length. If you select Same as first input, these characteristics will match those of the first input to the block. If you select User-defined, the **Output word length** and **Output fraction length** parameters become visible.

Output word length
> Specify the word length, in bits, of the output. This parameter is only visible if User-defined is specified for the **Fixed-point output attributes** parameter.

Output fraction length
> Specify the fraction length, in bits, of the output. This parameter is only visible if User-defined is specified for the **Fixed-point output attributes** parameter.

Round integer calculations toward
> Select the rounding mode for fixed-point operations.

Saturate on integer overflow
> If selected, overflows saturate.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point
- 8-, 16-, and 32-bit signed integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

DSP Gain	Signal Processing Blockset
DSP Sum	Signal Processing Blockset
Product	Simulink

DSP Sum

Purpose   Add two inputs

Library   dspobslib

Description
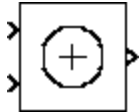
> **Note** The DSP Sum block is still supported but is likely to be obsoleted
> in a future release. We strongly recommend replacing this block with
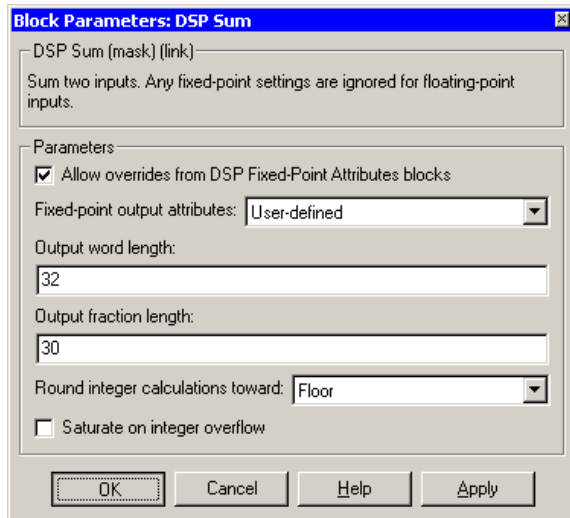> the Simulink Sum block.

The DSP Sum block is a masked version of the Simulink Sum block.
This block adds two inputs.

The inputs to the DSP Sum block can be scalar, vector, or matrix.
Either both inputs must have the same dimensions, or at least one of
the inputs must be a scalar:

- If one input is a scalar, it is added to each element of the other input.

- If both inputs are vectors or a matrices, their corresponding elements
  are added together.

The DSP Sum block accepts real and complex floating-point and
fixed-point inputs.

**Dialog
Box**

DSP Sum (mask) (link)

Sum two inputs. Any fixed-point settings are ignored for floating-point inputs.

Parameters

☑ Allow overrides from DSP Fixed-Point Attributes blocks

Fixed-point output attributes: User-defined

Output word length:

32

Output fraction length:

30

Round integer calculations toward: Floor

☐ Saturate on integer overflow

OK   Cancel   Help   Apply

Allow overrides from DSP Fixed-Point Attributes blocks

If you select this parameter, fixed-point data types for this block may be set by DSP Fixed-Point Attributes (DFPA) blocks in your model. If this parameter is unselected, the data types are always set by the parameters in the block mask.

When a DSP Sum block is overridden by a DFPA block, the output data type of the DSP Sum block is controlled by the **Accumulator** attribute. The **Output** attribute of the DFPA block is ignored for DSP Sum blocks.

Fixed-point output attributes

Choose how you will specify the output word length and fraction length. If you select Same as first input, these characteristics will match those of the first input to the block. If you select User-defined, the **Output word length** and **Output fraction length** parameters become visible.

DSP Sum

Output word length
Specify the word length, in bits, of the output. This parameter is only visible if User-defined is specified for the **Fixed-point output attributes** parameter.

Output fraction length
Specify the fraction length, in bits, of the output. This parameter is only visible if User-defined is specified for the **Fixed-point output attributes** parameter.

Round integer calculations toward
Select the rounding mode for fixed-point operations.

Saturate on integer overflow
If selected, overflows saturate.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point
- 8-, 16-, and 32-bit signed integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

DSP Gain	Signal Processing Blockset
DSP Product	Signal Processing Blockset
Sum	Simulink

Purpose   Compute discrete wavelet transform (DWT) of input

Library   Transforms

Description

> **Note** The DWT block is the same as the Dyadic Analysis Filter Bank
> block in the Multirate Filters library, but with different default settings.
> See the Dyadic Analysis Filter Bank block reference page for more
> information on how to use the block.

The DWT block computes the discrete wavelet transform (DWT) of each
column of a frame-based input. By default, the output is a sample-based
vector or matrix with the same dimensions as the input. Each column
of the output is the DWT of the corresponding input column.

You must install the Wavelet Toolbox for the block to automatically
design wavelet-based filters to compute the DWT. Otherwise, you must
specify your own lowpass and highpass FIR filters by setting the **Filter**
parameter to User defined.

For the same input, the DWT block and the dwt function, in the Wavelet
Toolbox, do not produce the same results. Because the blockset is
designed for real-time implementation and the toolbox is designed for
analysis, the products handle boundary conditions and filter states
differently. To make the output of the DWT block match the output of
the dwt function, complete the following steps:

1 For the dwt function, set the boundary condition to zero-padding by
typing dwtmode('zpd') at the MATLAB command prompt.

2 To match the latency of the DWT block, which is implemented using
FIR filters, add zeros to the input of the dwt function. The number of
zeros you add must be equal to the half the filter length.

For detailed information about how to use this block, see the Dyadic
Analysis Filter Bank block reference page.

DWT

Examples   See "Examples" on page 10-364 in the Dyadic Analysis Filter Bank block reference page.

See Also

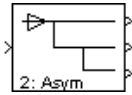|	
Dyadic Analysis Filter Bank	Signal Processing Blockset
IDWT	Signal Processing Blockset
dwt	Wavelet Toolbox

Purpose   Decompose signals into subbands with smaller bandwidths and slower
                    sample rates

Library   Filtering / Multirate Filters

Description

> **Note** This block decomposes frame-based signals with frame size
> a multiple of 2^n into either n+1 or 2^n subbands. To decompose
> sample-based signals or frame-based signals of different sizes, use the
> Two-Channel Analysis Subband Filter block. (You can connect multiple
> copies of the Two-Channel Analysis Subband Filter block to create a
> multilevel dyadic analysis filter bank.)

The Dyadic Analysis Filter Bank block decomposes a broadband signal
into a collection of subbands with smaller bandwidths and slower
sample rates. The block uses a series of highpass and lowpass FIR filters
to repeatedly divide the input frequency range, as illustrated in the
figure n-Level Asymmetric Dyadic Analysis Filter Bank on page 3-67.

You can specify the filter bank's highpass and lowpass filters by
providing vectors of filter coefficients. If you install the Wavelet Toolbox,
you can also specify wavelet-based filters by selecting a wavelet from the
Filter parameter. You must set the filter bank structure to asymmetric
or symmetric, and specify the number of levels in the filter bank.

Input Requirements

- Input can be a frame-based vector or frame-based matrix.

- The input frame size must be a multiple of 2^n, where n is the number
  of filter bank levels. For example, a frame size of 16 would be
  appropriate for a three-level tree (16 is a multiple of 2^3).

- The block always operates along the columns of the inputs.

For an illustration of why the above input requirements exist, see the
figure Outputs of a 3-Level Asymmetric Dyadic Analysis Filter Bank
on page 10-361.

Output Characteristics

The output characteristics vary depending on the block's parameter settings, as summarized in the following list and figure:

- **Number of levels** parameter set to n
- **Tree structure** parameter setting:
  - Asymmetric — Block produces $n+1$ output subbands
  - Symmetric — Block produces 2^n output subbands
- **Output** parameter setting can be Multiple ports or Single port. The following figure illustrates the difference between the two settings for a 3-level asymmetric dyadic analysis filter bank. For an explanation of the illustrated output characteristics, see the table Output Characteristics for an n-Level Dyadic Analysis Filter Bank on page 10-362.

For more information about the filter bank levels and structures, see "Dyadic Analysis Filter Banks" on page 3-66.
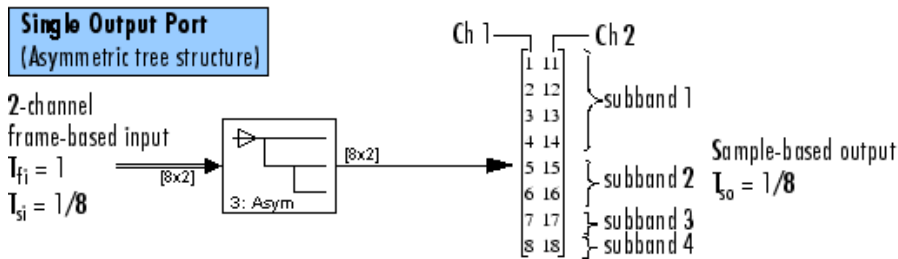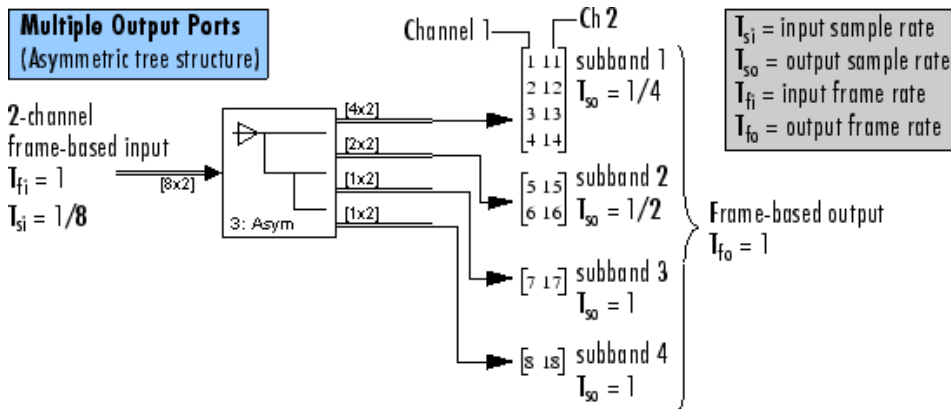
Outputs of a 3-Level Asymmetric Dyadic Analysis Filter Bank

The following table summarizes the different output characteristics of the block when it is set to output from single or multiple ports.

Dyadic Analysis Filter Bank

Output Characteristics for an n-Level Dyadic Analysis Filter Bank

	Single Output Port	**Multiple Output Ports**
Output Description	Block concatenates all the subbands into one vector or matrix, and outputs the concatenated subbands from a single output port. Each output column contains subbands of the corresponding input channel.	Block outputs each subband from a separate output port. The topmost port outputs the subband with the highest frequencies. Each output column contains a subband for the corresponding input channel.
Output Frame Status	Sample-based	Frame-based
Output Frame Rate	*Not applicable*	Same as input frame rate (However, the output frame sizes can vary, so the output sample rates can vary.)
Output Dimensions (Frame Size)	Same number of rows and columns as the input.	The output has the same number of columns as the input. The number of output rows is the output frame size. For an input with frame size M_i output y_k has frame size $M_{o,k}$: • Symmetric — All outputs have the frame size, $M_i / 2^n$. • Asymmetric — The frame size of each output (except the last) is half that of the output from the previous level. The outputs from the last two output ports have the same frame size since they originate from the same level in the filter bank.

	Single Output Port	Multiple Output Ports
		$$M_{o,k} = \begin{cases} M_i/2^k & (1 \le k \le n) \\ M_i/2^n & (k = n+1) \end{cases}$$
Output Sample Rate	Same as input sample rate.	Though the outputs have the same frame rate as the input, they have different frame sizes than the input. Thus, the output sample rates, $F_{so,k}$, are different from the input sample rate, F_{si}: • Symmetric — All outputs have the sample rate $F_{si} / 2^n$. • Asymmetric — $$F_{so,k} = \begin{cases} F_{si}/2^k & (1 \le k \le n) \\ F_{si}/2^n & (k = n+1) \end{cases}$$

Filter Bank Filters

You must specify the highpass and lowpass filters in the filter bank by setting the **Filter** parameter to one of the following options:

• User defined — Allows you to explicitly specify the filters with two vectors of filter coefficients in the **Lowpass FIR filter coefficients** and **Highpass FIR filter coefficients** parameters. The block uses the same lowpass and highpass filters throughout the filter bank. The two filters should be halfband filters, where each filter passes the frequency band that the other filter stops.

• Wavelet such as Biorthogonal or Daubechies — The block uses the specified wavelet to construct the lowpass and highpass filters using the Wavelet Toolbox function, wfilters. Depending on the wavelet, the block might enable either the **Wavelet order** or **Filter order**

Dyadic Analysis Filter Bank

[synthesis / analysis] parameter. (The latter parameter allows you to specify different wavelet orders for the analysis and synthesis filter stages.) You must install the Wavelet Toolbox to use wavelets.

Specifying Filters with the Filter Parameter and Related Parameters

Filter	Sample Setting for Related Filter Specification Parameters	Corresponding Wavelet Function Syntax
User-defined	Filters based on Daubechies wavelets with wavelet order 3: - **Highpass FIR filter coefficients =** `[-0.3327 0.8069 -0.4599 -0.1350 0.0854 0.0352]` - **Lowpass FIR filter coefficients =** `[0.0352 -0.0854 -0.1350 0.4599 0.8069 0.3327]`	None
Haar	None	`wfilters('haar')`
Daubechies	**Wavelet order =** 4	`wfilters('db4')`
Symlets	**Wavelet order =** 3	`wfilters('sym3')`
Coiflets	**Wavelet order =** 1	`wfilters('coif1')`
Biorthogonal	**Filter order [synthesis / analysis] =** [3/1]	`wfilters('bior3.1')`
Reverse Biorthogonal	**Filter order [synthesis / analysis] =** [3/1]	`wfilters('rbio3.1')`
Discrete Meyer	None	`wfilters('dmey')`

Examples

Wavelets

The primary application for dyadic analysis filter banks and dyadic synthesis filter banks, is coding for data compression using wavelets.

At the transmitting end, the output of the dyadic analysis filter bank is fed to a lossy compression scheme, which typically assigns the number of bits for each filter bank output in proportion to the relative energy in that frequency band. This represents the more powerful signal components by a greater number of bits than the less powerful signal components.
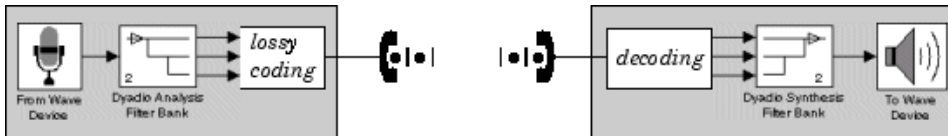
At the receiving end, the transmission is decoded and fed to a dyadic synthesis filter bank to reconstruct the original signal. The filter coefficients of the complementary analysis and synthesis stages are designed to cancel aliasing introduced by the filtering and resampling.

Demos

See the following Signal Processing Blockset demos, which use the Dyadic Analysis Filter Bank block:

- Multi-level PR filter bank

- Denoising

- Wavelet transmultiplexer (WTM)

Dyadic Analysis Filter Bank

> **Note** To see the version of the demos that use the Dyadic Analysis Filter Bank and Dyadic Synthesis Filter Bank blocks, click the **Frame-Based Demo** button in the demos.
>
> Open the demos using one of the following methods:
>
> - Click the above links in the MATLAB Help browser (*not* in a Web browser).
> - Type `demo blockset dsp` at the MATLAB command line, and look in the `Wavelets` directory.

Dialog Box

The parameters displayed in the block dialog vary depending on the setting of the **Filter** parameter. Only some of the parameters described below are visible in the dialog box at any one time.
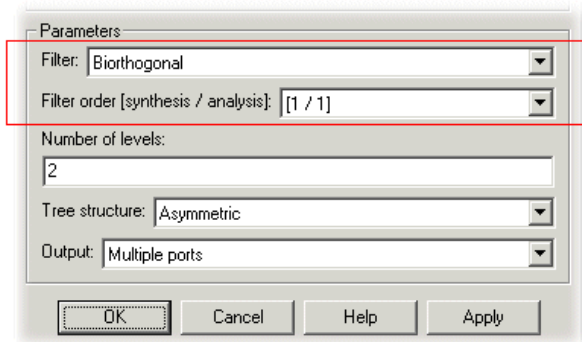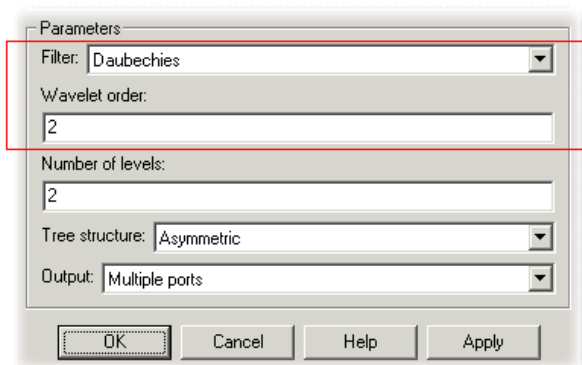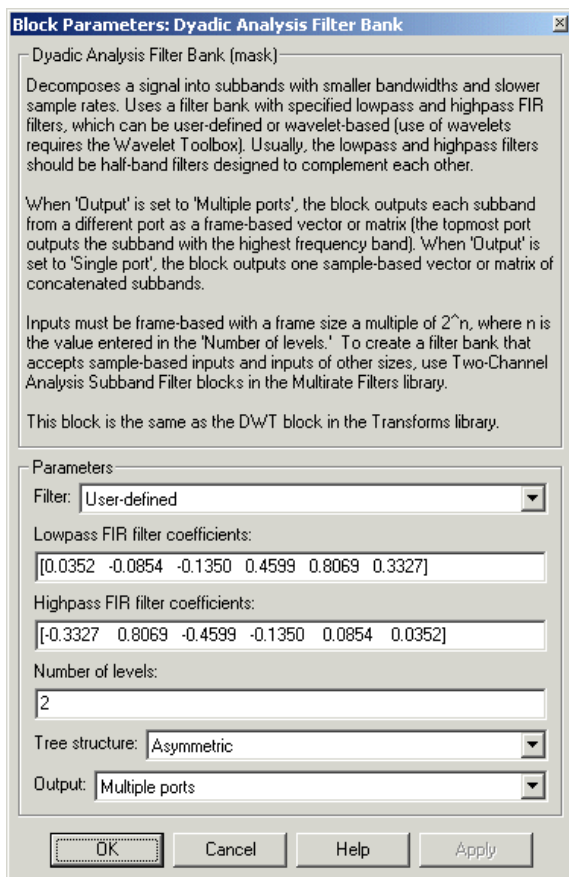
Filter

The type of filter used to determine the high- and low-pass FIR filters in the dyadic analysis filter bank:

Select User defined to explicitly specify the filter coefficients in the **Lowpass FIR filter coefficients** and **Highpass FIR filter coefficients** parameters.

Dyadic Analysis Filter Bank

Select a wavelet such as `Biorthogonal` or `Daubechies` to specify a wavelet-based filter. The block uses the Wavelet Toolbox function, `wfilters`, to construct the filters. Extra parameters such as **Wavelet order** or **Filter order [synthesis / analysis]** might become enabled. For a list of the supported wavelets, see Specifying Filters with the Filter Parameter and Related Parameters on page 10-364.

Lowpass FIR filter coefficients

A vector of filter coefficients (descending powers of z) that specifies coefficients used by all the lowpass filters in the filter bank. This parameter is enabled when you set **Filter** to `User defined`. The lowpass filter should be a half-band filter that passes the frequency band stopped by the filter specified in the **Highpass FIR filter coefficients** parameter. The default values of this parameter specify a filter based on aDaubechies wavelet with wavelet order 3.

Highpass FIR filter coefficients

A vector of filter coefficients (descending powers of z) that specifies coefficients used by all the highpass filters in the filter bank. This parameter is enabled when you set **Filter** to `User defined`. The highpass filter should be a half-band filter that passes the frequency band stopped by the filter specified in the **Lowpass FIR filter coefficients** parameter. The default values of this parameter specify a filter based on a Daubechies wavelet with wavelet order 3.

Wavelet order

The order of the wavelet selected in the **Filter** parameter. This parameter is enabled only when you set **Filter** to certain types of wavelets, as shown in the Specifying Filters with the Filter Parameter and Related Parameters table.

Filter order [synthesis / analysis]

The order of the wavelet for the synthesis and analysis filter stages. For example, when you set the **Filter** parameter to `Biorthogonal` and set the **Filter order [synthesis / analysis]** parameter to [2 / 6], the block calls the `wfilters` function with

input argument `'bior2.6'`. This parameter is enabled only when you set **Filter** to certain types of wavelets, as shown in Specifying Filters with the Filter Parameter and Related Parameters on page 10-364.

Number of levels

The number of filter bank levels. An n-level asymmetric structure has $n+1$ outputs, and an n-level symmetric structure has 2^n outputs, as shown in the figures n-Level Asymmetric Dyadic Analysis Filter Bank on page 3-67 and n-Level Symmetric Dyadic Analysis Filter Bank on page 3-68. The block's icon displays the value of this parameter in the lower-left corner.

Tree structure

The structure of the filter bank: `Asymmetric`, or `Symmetric`. See the figures n-Level Asymmetric Dyadic Analysis Filter Bank on page 3-67 and n-Level Symmetric Dyadic Analysis Filter Bank on page 3-68.

Output

Set to `Multiple ports` to output each output subband on a separate port (the topmost port outputs the subband with the highest frequency band). Set to `Single port` to concatenate the subbands into one vector or matrix and output the concatenated subbands on a single port. For more information, see "Output Characteristics" on page 10-360.

References

Fliege, N. J. *Multirate Digital Signal Processing: Multirate Systems, Filter Banks, Wavelets*. West Sussex, England: John Wiley & Sons, 1994.

Strang, G. and T. Nguyen. *Wavelets and Filter Banks*. Wellesley, MA: Wellesley-Cambridge Press, 1996.

Vaidyanathan, P. P. *Multirate Systems and Filter Banks*. Englewood Cliffs, NJ: Prentice Hall, 1993.

Dyadic Analysis Filter Bank

Supported Data Types

- Double-precision floating point
- Single-precision floating point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Dyadic Synthesis Filter Bank	Signal Processing Blockset
Two-Channel Analysis Subband Filter	Signal Processing Blockset

Purpose   Reconstruct signals from subbands with smaller bandwidths and slower sample rates

Library   Filtering / Multirate Filters

Description
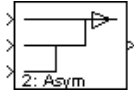
Note This block always outputs frame-based signals, and its inputs must be of certain sizes. To get sample-based outputs or to use input subbands that do not fit the criteria of this block, use the Two-Channel Synthesis Subband Filter block. (You can connect multiple copies of the Two-Channel Synthesis Subband Filter block to create a multilevel dyadic synthesis filter bank.)

The Dyadic Synthesis Filter Bank block reconstructs a signal decomposed by the Dyadic Analysis Filter Bank block. The block takes in subbands of a signal, and uses them to reconstruct the signal by using a series of highpass and lowpass FIR filters as illustrated in the figure n-Level Asymmetric Dyadic Synthesis Filter Bank on page 3-71. The reconstructed signal has a wider bandwidth and faster sample rate than the input subbands.

You can specify the filter bank's highpass and lowpass filters by providing vectors of filter coefficients. If you install the Wavelet Toolbox, you can also specify wavelet-based filters by selecting a wavelet from the **Filter** parameter.

Dyadic Synthesis Filter Bank

Note 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.

This block automatically computes wavelet-based perfect reconstruction filters when the wavelet selection in the **Filter** parameter of this block is the *same* as the **Filter** parameter setting of the corresponding Dyadic Analysis Filter Bank block. The use of wavelets requires the Wavelet Toolbox. To learn how to design your own perfect reconstruction filters, see "References" on page 10-381.

Input Requirements

The inputs to this block are usually the outputs of a Dyadic Analysis Filter Bank block. Since the Dyadic Analysis Filter Bank block can output from either a single port or multiple ports, the Dyadic Synthesis Filter Bank block accepts inputs to either a single port or multiple ports.

The **Input** parameter sets whether the block accepts inputs from a single port or multiple ports, and thus determines the input requirements, as summarized in the following lists and figure.

Note Any output of a Dyadic Analysis Filter Bank block whose parameter settings match the corresponding settings of this block is a valid input to this block. For example, the setting of the Dyadic Analysis Filter Bank block parameter, **Output**, must be the same as this block's **Input** parameter (`Single port` or `Multiple ports`).

Valid Inputs for Input Set to Single Port

- Inputs must be sample-based vectors or sample-based matrices of concatenated subbands.

- Each input column contains the subbands for an independent signal.

- Upper input rows contain the high-frequency subbands, and the lower rows contain the low-frequency subbands.

Valid Inputs for Input Set to Multiple Ports

- Inputs must be a frame-based vector or frame-based matrix for each subband, each of which is input to a separate input port.

- The columns of each input contains a subband for an independent signal.

- The input to the topmost input port is the subband containing the highest frequencies, and the input to the bottommost port is the subband containing the lowest frequencies.

Dyadic Synthesis Filter Bank
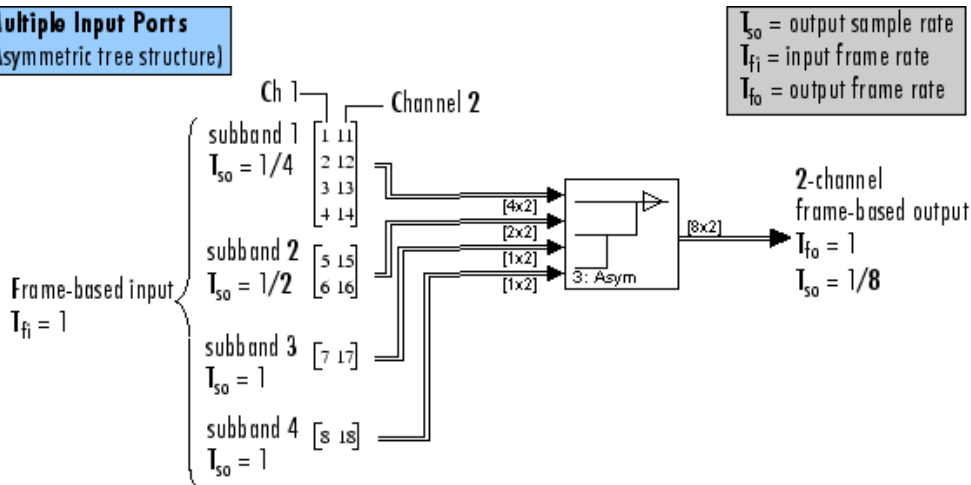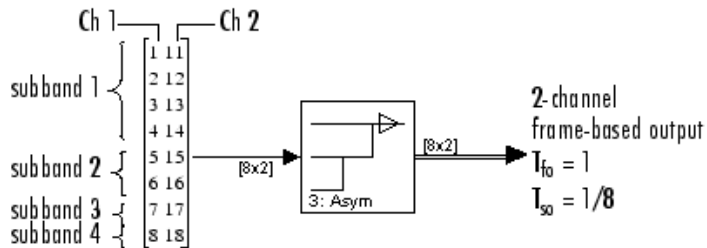
Valid Inputs to a 3-Level Asymmetric Dyadic Synthesis Filter Bank

For general information about the filter banks, see "Dyadic Synthesis Filter Banks" on page 3-70.

Output Characteristics

The following table summarizes the output characteristics for both frame-based inputs, and concatenated subband inputs. For an illustration of why the output characteristics exist, see the figure Valid Inputs to a 3-Level Asymmetric Dyadic Synthesis Filter Bank on page 10-374.

	Frame-Based Inputs (Input = Multiple ports)	Concatenated Subband Inputs (Input = Single port)
Output Frame Status	Outputs are always frame based regardless of the input frame status. Each output column is an independent channel, reconstructed from the corresponding channel in the inputs.	
Output Frame Rate	Same as the input frame rate.	Same as the input rate (the rate of the concatenated subband inputs).

Output Frame Dimensions	• The output has the same number of columns as the inputs.
	
	• The number of output rows depends on the tree structure of the filter bank:
	
	  ▪ **Asymmetric** — The number of output rows is twice the number of rows in the input to the topmost input port.
	
	  ▪ **Symmetric** — The number of output rows is the product of the number of input ports and the number of rows in an input to any input port.
		The output has the same number of rows and columns as the input.

For general information about the filter banks, see "Dyadic Synthesis Filter Banks" on page 3-70.

Filter Bank Filters

You must specify the highpass and lowpass filters in the filter bank by setting the **Filter** parameter to one of the following options:

• User defined — Allows you to explicitly specify the filters with two vectors of filter coefficients in the **Lowpass FIR filter coefficients** and **Highpass FIR filter coefficients** parameters. The block uses

the same lowpass and highpass filters throughout the filter bank. The two filters should be halfband filters, where each filter passes the frequency band that the other filter stops. To use this block to perfectly reconstruct a signal decomposed by a Dyadic Analysis Filter Bank block, the filters in this block *must* be designed to perfectly reconstruct the outputs of the analysis filter bank. To learn how to design your own perfect reconstruction filters, see "References" on page 10-381.

- Wavelet such as `Biorthogonal` or `Daubechies` — The block uses the specified wavelet to construct the lowpass and highpass filters using the Wavelet Toolbox function, `wfilters`. Depending on the wavelet, the block might enable either the **Wavelet order** or **Filter order [synthesis / analysis]** parameter. (The latter parameter allows you to specify different wavelet orders for the analysis and synthesis filter stages.) To use this block to reconstruct a signal decomposed by a Dyadic Analysis Filter Bank block, you must set both blocks to use the same wavelets with the same order. You must install the Wavelet Toolbox to use wavelets.

Specifying Filters with the Filter Parameter and Related Parameters

Filter	Sample Setting for Related Filter Specification Parameters	Corresponding Wavelet Function Syntax
User-defined	Filters based on Daubechies wavelets with wavelet order 3: - **Lowpass FIR filter coefficients =** `[0.0352 -0.0854 -0.1350 0.4599 0.8069 0.3327]` - **Highpass FIR filter coefficients =** `[-0.3327 0.8069 -0.4599 -0.1350 0.0854 0.0352]`	None

Filter	Sample Setting for Related Filter Specification Parameters	Corresponding Wavelet Function Syntax
Haar	None	`wfilters('haar')`
Daubechies	**Wavelet order =** 4	`wfilters('db4')`
Symlets	**Wavelet order =** 3	`wfilters('sym3')`
Coiflets	**Wavelet order =** 1	`wfilters('coif1')`
Biorthogonal	**Filter order [synthesis / analysis] =** [3/1]	`wfilters('bior3.1')`
Reverse Biorthogonal	**Filter order [synthesis / analysis] =** [3/1]	`wfilters('rbio3.1')`
Discrete Meyer	None	`wfilters('dmey')`

Examples   See "Examples" on page 10-364 in the Dyadic Analysis Filter Bank block reference.

Dialog Box   The parameters displayed in the block dialog vary depending on the setting of the **Filter** parameter. Only some of the parameters described below are visible in the dialog box at any one time.

Dyadic Synthesis Filter Bank
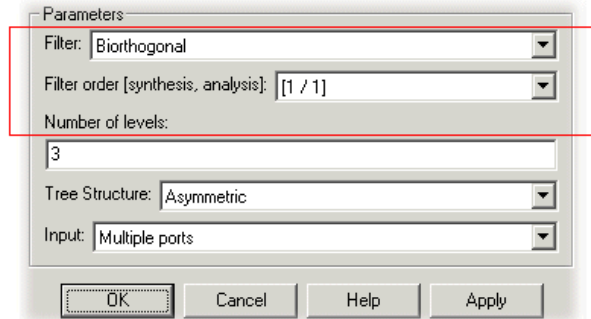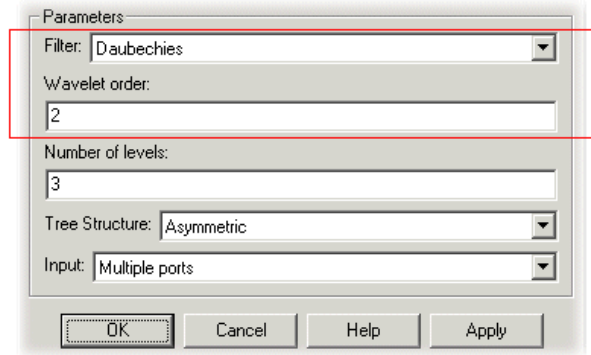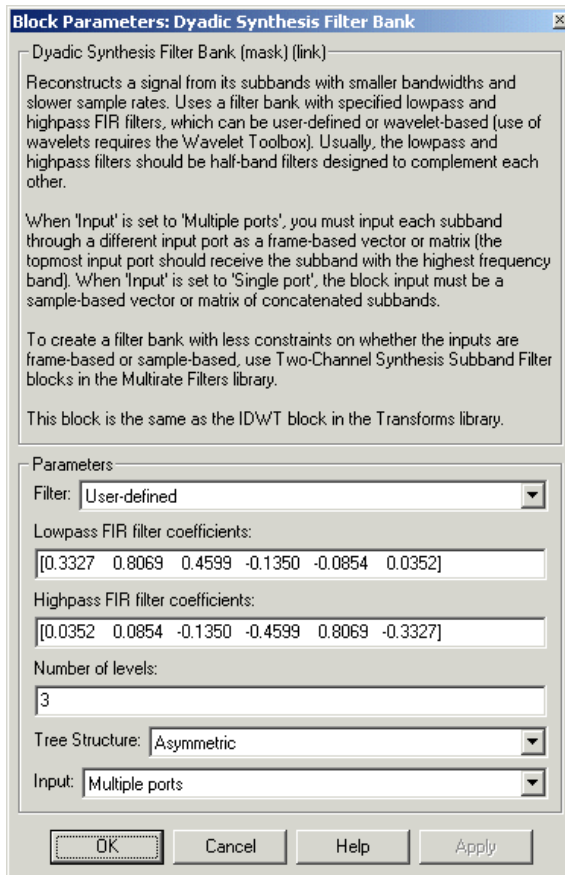
Note To use this block to reconstruct a signal decomposed by a Dyadic Analysis Filter Bank block, all the parameters in this block must be the same as the corresponding parameters in the Dyadic Analysis Filter Bank block (except the **Lowpass FIR filter coefficients** and **Highpass FIR filter coefficients**; see the descriptions of these parameters).

Filter

The type of filter used to determine the high- and low-pass FIR filters in the dyadic synthesis filter bank:

- Select `User defined` to explicitly specify the filter coefficients in the **Lowpass FIR filter coefficients** and **Highpass FIR filter coefficients** parameters.

- Select a wavelet such as `Biorthogonal` or `Daubechies` to specify a wavelet-based filter. The block uses the Wavelet Toolbox function, `wfilters`, to construct the filters. Extra parameters such as **Wavelet order** or **Filter order [synthesis / analysis]** might become enabled. For a list of the supported wavelets, see the table Specifying Filters with the Filter Parameter and Related Parameters on page 10-376.

Lowpass FIR filter coefficients

A vector of filter coefficients (descending powers of z) that specifies coefficients used by all the lowpass filters in the filter bank. This parameter is enabled when you set **Filter** to `User defined`. The lowpass filter should be a half-band filter that passes the frequency band stopped by the filter specified in the **Highpass FIR filter coefficients** parameter. To perfectly reconstruct a signal decomposed by the Dyadic Analysis Filter Bank, the filters in this block *must* be designed to perfectly reconstruct the outputs of the analysis filter bank. Otherwise, the reconstruction will not be perfect. The default values of this parameter specify a perfect reconstruction filter for the default settings of the Dyadic Analysis Filter Bank (based on a Daubechies wavelet with wavelet order 3).

Highpass FIR filter coefficients

A vector of filter coefficients (descending powers of z) that specifies coefficients used by all the highpass filters in the filter bank. This parameter is enabled when you set **Filter** to `User defined`. The highpass filter should be a half-band filter that passes the frequency band stopped by the filter specified in the **Lowpass FIR filter coefficients** parameter. To perfectly reconstruct a signal decomposed by the Dyadic Analysis Filter Bank, the filters in this block *must* be designed to perfectly reconstruct the outputs

of the analysis filter bank. Otherwise, the reconstruction will not be perfect. The default values of this parameter specify a perfect reconstruction filter for the default settings of the Dyadic Analysis Filter Bank (based on a Daubechies wavelet with wavelet order 3).

Wavelet order

The order of the wavelet selected in the **Filter** parameter. This parameter is enabled only when you set **Filter** to certain types of wavelets, as shown in the table Specifying Filters with the Filter Parameter and Related Parameters on page 10-376.

Filter order [synthesis / analysis]

The order of the wavelet for the synthesis and analysis filter stages. For example, when you set the **Filter** parameter to **Biorthogonal** and set the **Filter order [synthesis / analysis]** parameter to [2 / 6], the block calls the wfilters function with input argument 'bior2.6'. This parameter is enabled only when you set **Filter** to certain types of wavelets, as shown in Specifying Filters with the Filter Parameter and Related Parameters on page 10-376.

Number of levels

The number of filter bank levels. An n-level asymmetric structure has $n+1$ outputs, and an n-level symmetric structure has 2^n outputs, as shown in n-Level Asymmetric Dyadic Synthesis Filter Bank on page 3-71 and n-Level Symmetric Dyadic Synthesis Filter Bank on page 3-72.

Tree structure

The structure of the filter bank: Asymmetric, or Symmetric. See the figures n-Level Asymmetric Dyadic Synthesis Filter Bank on page 3-71 and n-Level Symmetric Dyadic Synthesis Filter Bank on page 3-72.

Input

Set to Multiple ports to accept each input subband at a separate port (the topmost port accepts the subband with the highest frequency band). Set to Single port to accept one vector or

matrix of concatenated subbands at a single port. For more information, see "Input Requirements" on page 10-372.

References Fliege, N. J. *Multirate Digital Signal Processing: Multirate Systems, Filter Banks, Wavelets*. West Sussex, England: John Wiley & Sons, 1994.

Strang, G. and T. Nguyen. *Wavelets and Filter Banks*. Wellesley, MA: Wellesley-Cambridge Press, 1996.

Vaidyanathan, P. P. *Multirate Systems and Filter Banks*. Englewood Cliffs, NJ: Prentice Hall, 1993.

Supported Data Types

- Double-precision floating point
- Single-precision floating point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Dyadic Analysis Filter Bank	Signal Processing Blockset
Two-Channel Synthesis Subband Filter	Signal Processing Blockset

See "Multirate Filters" on page 3-66 for related information.

Edge Detector

Purpose   Detect transition from zero to a nonzero value

Library   Signal Management / Switches and Counters
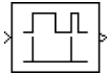
Description The Edge Detector block generates an impulse (the value 1) in a given output channel when the corresponding channel of the input transitions from zero to a nonzero value. Otherwise, the block generates zeros in each channel.

The output has the same dimension and sample rate as the input. When the input is frame based, the output is frame based; otherwise, the output is sample based. For frame-based input, an edge that is split across two consecutive frames (that is, a zero at the bottom of the first frame, and a nonzero value at the top of the following frame) is counted in the frame that contains the nonzero value.

Examples   In the model below, the Edge Detector block locates the edges (zero to nonzero transitions) in a two-channel frame-based input with frame size 3. The two input channels are horizontally concatenated with the two output channels to create the four-channel workspace variable dsp_examples_yout.
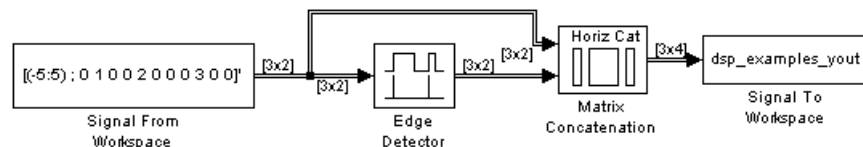
Adjust the block parameters as described below. (Use the default settings for the To Workspace block.)

• Set the Signal From Workspace block parameters as follows:

  ▪ **Signal** = [(-5:5) ; 0 1 0 0 2 0 0 0 3 0 0]'

  ▪ **Sample time** = 1

  ▪ **Samples per frame** = 3

- Set the Matrix Concatenation block parameters as follows:

  - **Number of inputs** = 2
  - **Concatenation method** = Horizontal

As shown below, the block finds edges at sample 7 in channel 1, and at samples 2, 5, and 9 in channel 2.
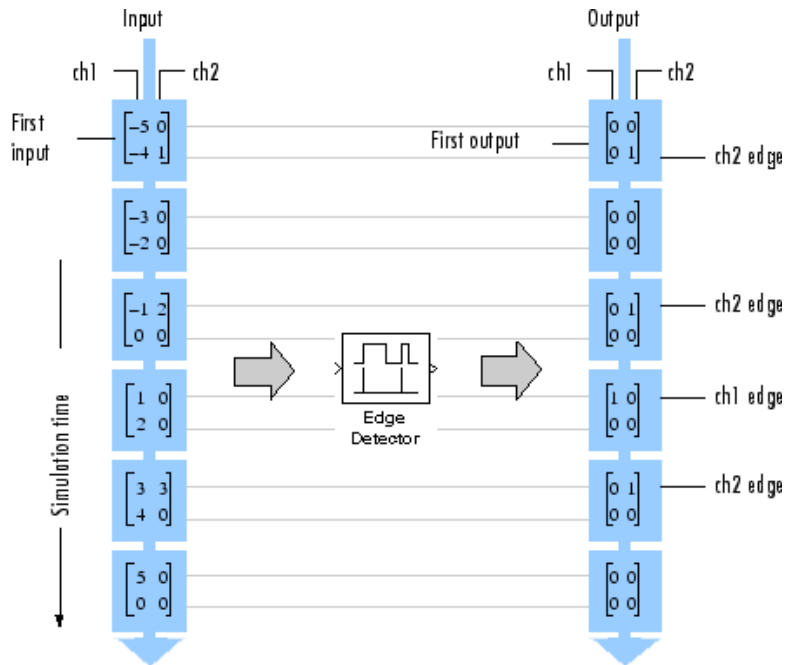
Dialog Box
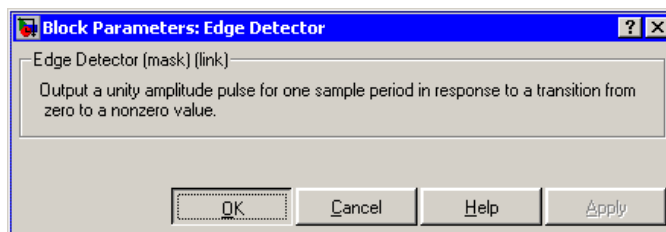
Edge Detector

Supported Data Types

- Double-precision floating point

- Single-precision floating point

- Fixed point (signed and unsigned)

- Custom data types

- Boolean — The block might output Boolean values depending on the input data type, and whether Boolean support is enabled or disabled, as described in "Effects of Enabling and Disabling Boolean Support" on page 7-17. To learn how to disable Boolean output support, see "Steps to Disabling Boolean Support" on page 7-18.

- 8-, 16-, and 32-bit signed integers

- 8-, 16-, and 32-bit unsigned integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

|	
Counter	Signal Processing Blockset
Event-Count Comparator	Signal Processing Blockset

Purpose   Detect threshold crossing of accumulated nonzero inputs

Library   Signal Management / Switches and Counters

Description   The Event-Count Comparator block records the number of nonzero
inputs to the Data port during the period that the block is enabled by a
high signal (the value 1) at the Int port. Both inputs must be scalars;
the input to the Int port can be sample based or frame based. When
the input to the Data port is frame based, the output is frame based;
otherwise, the output is sample based.

When the number of accumulated nonzero inputs first equals the **Event
threshold** setting, the block waits one additional sample interval, and
then sets the output high (1). The block holds the output high until
recording is restarted by a low-to-high (0-to-1) transition at the Int port.

The Event-Count Comparator block accepts real and complex
floating-point and fixed-point inputs. However, because the block
has discrete state, it does not support constant or continuous sample
times. Therefore, at least one input or output port of the Event-Count
Comparator block must be connected to a block whose **Sample time**
parameter is discrete. The Event-Count Comparator block inherits this
non-infinite discrete sample time.

Examples   In the model below, the Event-Count Comparator block (**Event
threshold** = 3) detects two threshold crossings in the input to the Data
port, one at sample 4 and one at sample 12.
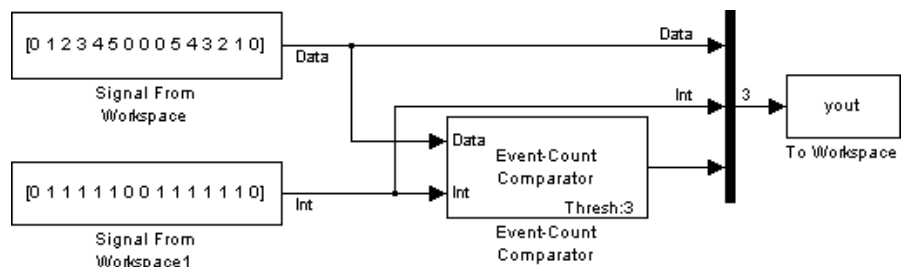
Event-Count Comparator

All inputs and outputs are multiplexed into the workspace variable yout, whose contents are shown in the figure below. The two left columns in the illustration show the inputs to the Data and Int ports, the center column shows the state of the block's internal counter, and the right column shows the block's output.
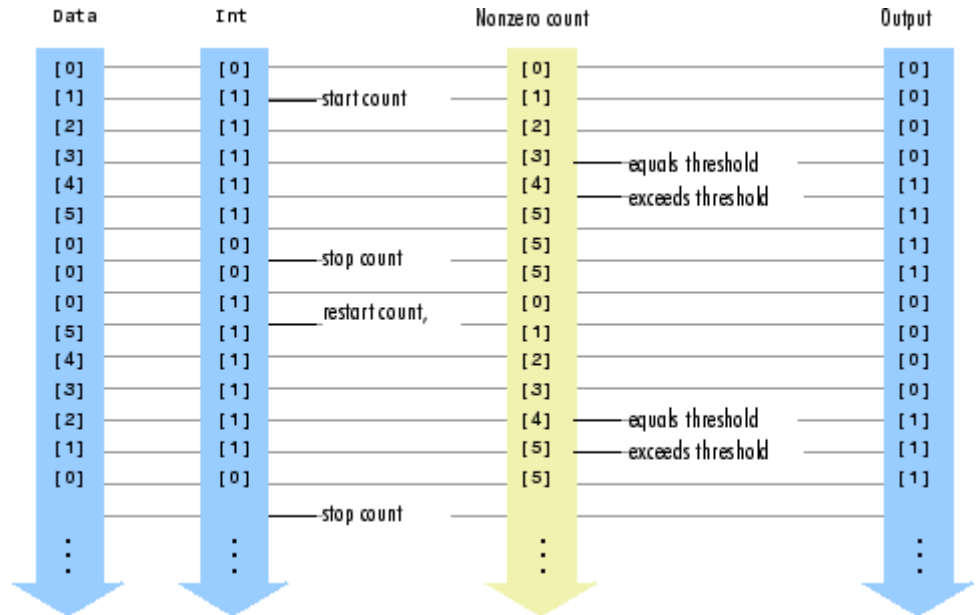
Dialog Box
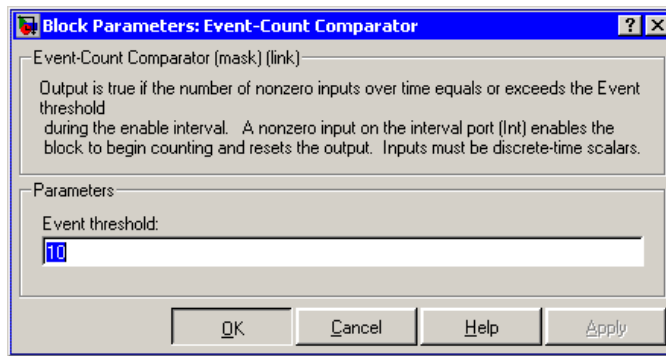
Event threshold

Specify the value against which to compare the number of nonzero inputs. Tunable.

Supported Data Types

- Double-precision floating point

- Single-precision floating point

- Fixed point (signed and unsigned)

- Custom data types

- Boolean

- 8-, 16-, and 32-bit signed integers

- 8-, 16-, and 32-bit unsigned integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

|	
Counter	Signal Processing Blockset
Edge Detector	Signal Processing Blockset

Extract Diagonal

Purpose　　　Extract main diagonal of input matrix

Library　　　Math Functions / Matrices and Linear Algebra / Matrix Operations

Description　　The Extract Diagonal block populates the 1-D output vector with the elements on the main diagonal of the *M*-by-*N* input matrix A.
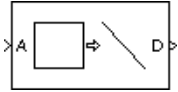
```
D = diag(A)     Equivalent MATLAB code
```

The output vector has length min(M,N), and is always sample based.
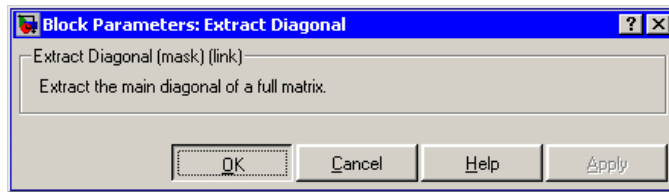
Dialog Box

```
Block Parameters: Extract Diagonal              ? X
─Extract Diagonal (mask) (link)──────────────────────
 Extract the main diagonal of a full matrix.

         [   OK   ]   [ Cancel ]   [  Help  ]   [ Apply ]
```

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point (signed and unsigned)
- Custom data types
- Boolean — Block outputs are always Boolean. To learn how to disable Boolean support, see "Steps to Disabling Boolean Support" on page 7-18.
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Constant Diagonal Matrix	Signal Processing Blockset
Create Diagonal Matrix	Signal Processing Blockset
Extract Triangular Matrix	Signal Processing Blockset
diag	MATLAB

Extract Triangular Matrix

Purpose   Extract lower or upper triangle from input matrices

Library   Math Functions / Matrices and Linear Algebra / Matrix Operations
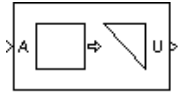
Description   The Extract Triangular Matrix block creates a triangular matrix output
from the upper or lower triangular elements of an M-by-N input matrix.
A length-M 1-D vector input is treated as an M-by-1 matrix.

The **Extract** parameter selects between the two components of the
input:
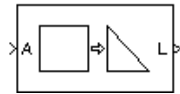
- Upper — Copies the elements on and above the main diagonal of the
  input matrix to an output matrix of the same size. The first *row* of
  the output matrix is therefore identical to the first *row* of the input
  matrix. The elements below the main diagonal of the output matrix
  are zero.

- Lower — Copies the elements on and below the main diagonal of the
  input matrix to an output matrix of the same size. The first *column*
  of the output matrix is therefore identical to the first *column* of the
  input matrix. The elements above the main diagonal of the output
  matrix are zero.

The output has the same frame status as the input.

Examples   The example below shows the extraction of upper and lower triangles
from a 5-by-3 input matrix.
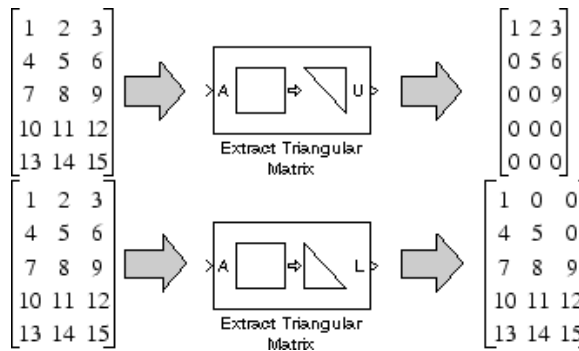
Dialog Box
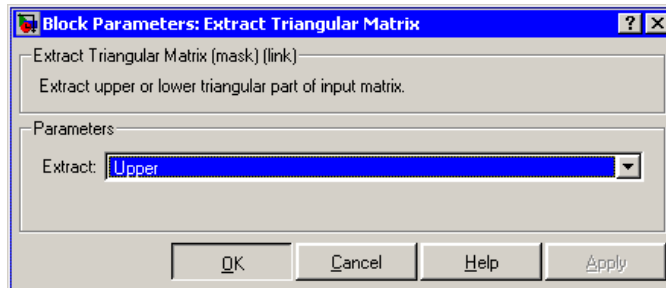
Extract

> The component of the matrix to copy to the output, upper triangle or lower triangle. Tunable.

Extract Triangular Matrix

Supported Data Types

Port	Supported Data Types
A	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Custom data types
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
U	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Custom data types
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
L	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Custom data types
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Autocorrelation LPC	Signal Processing Blockset
Cholesky Factorization	Signal Processing Blockset
Constant Diagonal Matrix	Signal Processing Blockset
Extract Diagonal	Signal Processing Blockset
Forward Substitution	Signal Processing Blockset
LDL Factorization	Signal Processing Blockset
LU Factorization	Signal Processing Blockset
tril	MATLAB
triu	MATLAB

Fast Block LMS Filter

Purpose
Compute filtered output, filter error, and filter weights for a given input and desired signal using the Fast Block LMS adaptive filter algorithm

Library
Filtering / Adaptive Filters

Description
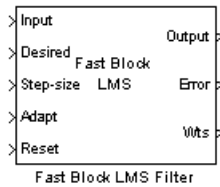
The Fast Block LMS Filter block implements an adaptive least mean-square (LMS) filter, where the adaptation of the filter weights occurs once for every block of data samples. The block estimates the filter weights, or coefficients, needed to convert the input signal into the desired signal. Connect the signal you want to filter to the Input port. This input signal can be a sample-based scalar or a single-channel frame-based signal. Connect the signal you want to model to the Desired port. The desired signal must have the same data type, frame status, complexity, and dimensions as the input signal. The Output port outputs the filtered input signal, which can be sample or frame based. The Error port outputs the result of subtracting the output signal from the desired signal.

The block calculates the filter weights using the Block LMS Filter equations. For more information, see Block LMS Filter. The Fast Block LMS Filter block implements the convolution operation involved in the calculations of the filtered output, y, and the weight update function in the frequency domain using the FFT algorithm used in the Overlap-Save FFT Filter block. See Overlap-Save FFT Filter for more information.

Use the **Filter length** parameter to specify the length of the filter weights vector.

The **Block size** parameter determines how many samples of the input signal are acquired before the filter weights are updated. The input frame length must be a multiple of the **Block size** parameter.

The **Step-size (mu)** parameter corresponds to μ in the equations. You can either specify a step-size using the input port, Step-size, or enter a value in the Block Parameters: Block LMS Filter dialog box.

Use the **Leakage factor (0 to 1)** parameter to specify the leakage factor, $0 < 1 - \mu\alpha \leq 1$, in the leaky LMS algorithm shown below.

$$\mathbf{w}(k) = (1 - \mu\alpha)\mathbf{w}(k-1) - f(\mathbf{u}(n), e(n), \mu)$$

Enter the initial filter weights, $\mathbf{w}(0)$, as a vector or a scalar in the **Initial value of filter weights** text box. When you enter a scalar, the block uses the scalar value to create a vector of filter weights. This vector has length equal to the filter length and all of its values are equal to the scalar value.

When you select the **Adapt port** check box, an Adapt port appears on the block. When the input to this port is nonzero, the block continuously updates the filter weights. When the input to this port is zero, the filter weights remain at their current values.

When you want to reset the value of the filter weights to their initial values, use the **Reset input** parameter. The block resets the filter weights whenever a reset event is detected at the Reset port. The reset signal rate must be the same rate as the data signal input.

From the **Reset input** list, select None to disable the Reset port. To enable the Reset port, select one of the following from the **Reset input** list:

- Rising edge — Triggers a reset operation when the Reset input does one of the following:

  - Rises from a negative value to a positive value or zero

  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)

- `Falling edge` — Triggers a reset operation when the Reset input does one of the following:

  - Falls from a positive value to a negative value or zero

  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero (see the following figure)
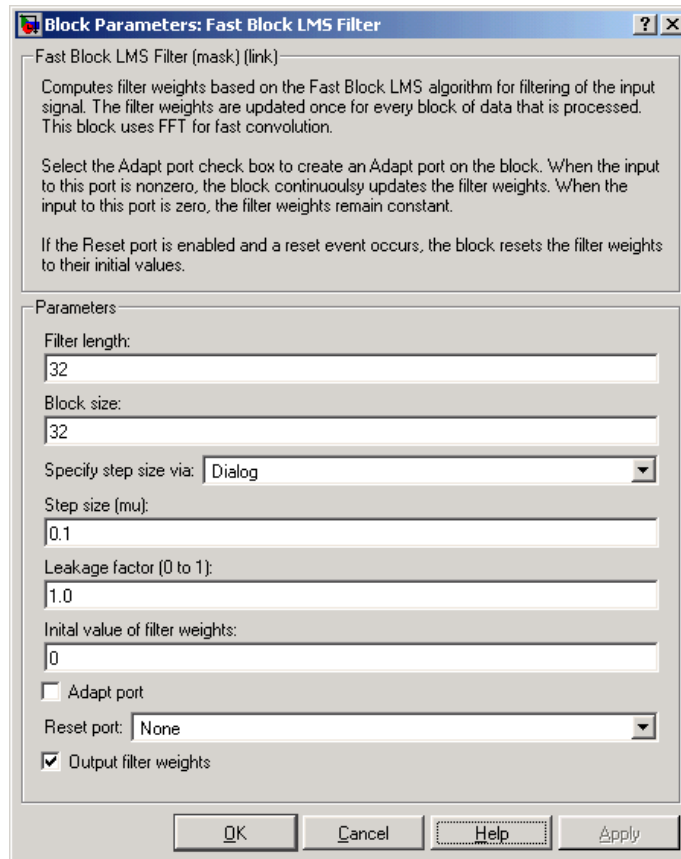

- `Either edge` — Triggers a reset operation when the Reset input is a `Rising edge` or `Falling edge` (as described above)

- `Non-zero sample` — Triggers a reset operation at each sample time that the Reset input is not zero

Note When running simulations in the Simulink `MultiTasking` mode, sample-based reset signals have a one-sample latency, and frame-based reset signals have one frame of latency. Thus, there is a one-sample or one-frame delay between the time the block detects a reset event, and when it applies the reset. For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" on page 2-56 and "Models with Multiple Sample Rates" in the Real-Time Workshop User's Guide documentation.

Select the **Output filter weights** check box to create a Wts port on the block. For each iteration, the block outputs the current updated filter weights from this port.

Fast Block LMS Filter

**Dialog
Box**

Block Parameters: Fast Block LMS Filter

Fast Block LMS Filter (mask) (link)

Computes filter weights based on the Fast Block LMS algorithm for filtering of the input signal. The filter weights are updated once for every block of data that is processed. This block uses FFT for fast convolution.

Select the Adapt port check box to create an Adapt port on the block. When the input to this port is nonzero, the block continuoulsy updates the filter weights. When the input to this port is zero, the filter weights remain constant.

If the Reset port is enabled and a reset event occurs, the block resets the filter weights to their initial values.

Parameters

Filter length:

32

Block size:

32

Specify step size via: Dialog

Step size (mu):

0.1

Leakage factor (0 to 1):

1.0

Inital value of filter weights:

0

☐ Adapt port

Reset port: None

☑ Output filter weights

OK   Cancel   Help   Apply

Filter length
>   Enter the length of the FIR filter weights vector. The sum of the
>   block size and the filter length must be a power of 2.

Block size
>   Enter the number of samples to acquire before the filter weights
>   are updated. The input frame length must be an integer multiple
>   of the block size. The sum of the block size and the filter length
>   must be a power of 2.

Specify step-size via

Select `Dialog` to enter a value for mu, or select `Input port` to specify mu using the Step-size input port.

Step-size (mu)

Enter the step-size. Tunable.

Leakage factor (0 to 1)

Enter the leakage factor, $0 < 1 - \mu\alpha \leq 1$. Tunable.

Initial value of filter weights

Specify the initial values of the FIR filter weights.

Adapt port

Select this check box to enable the Adapt input port.

Reset input

Select this check box to enable the Reset input port.

Output filter weights

Select this check box to export the filter weights from the Wts port.

References   Hayes, M.H. *Statistical Digital Signal Processing and Modeling.* New York: John Wiley & Sons, 1996.

Supported Data Types

- Double-precision floating point
- Single-precision floating point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Kalman Adaptive Filter	Signal Processing Blockset
LMS Filter	Signal Processing Blockset
RLS Filter	Signal Processing Blockset

Fast Block LMS Filter

Fast Block LMS Filter	Signal Processing Blockset
Overlap-Save FFT Filter	Signal Processing Blockset

Purpose   Compute fast Fourier transform (FFT) of input

Library   Transforms

Description   The FFT block computes the fast Fourier transform (FFT) of each
                   channel of an *M*-by-*N* or length-*M* input, *u*, where *M* must be a power of
                   two. To work with other input sizes, use the Zero Pad block to pad or
                   truncate the length-*M* dimension to a power-of-two length.

> FFT >

The output of the FFT block is equivalent to the MATLAB `fft` function:

```
y = fft(u)     % Equivalent MATLAB code
```

The *k*th entry of the *l*th output channel, $y(k, l)$, is equal to the *k*th point of
the *M*-point discrete Fourier transform (DFT) of the *l*th input channel:

$$y(k, l) = \sum_{m=1}^{M} u(m,l)e^{-j2\pi(m-1)(k-1)/M} \quad k = 1, ..., M$$

This block supports real and complex floating-point and fixed-point
inputs.

Input and Output Characteristics

The following table describes valid inputs to the FFT block, their
corresponding outputs, and the dimension along which the block
computes the DFT.

Valid Block Inputs • Real- or complex-valued • Must be in linear order • M must be a power of two.	**Dimension Along Which Block Computes DFT**	**Corresponding Block Output Characteristics** **Output port rate = input port rate**
Frame-based *M*-by-*N* matrix	Column	• Sample based • Complex valued • *M*-by-*N* matrix • Each output column contains the *M*-point DFT of the corresponding input channel in linear or bit-reversed order.
Sample-based *M*-by-*N* matrix, $M \neq 1$	Column	• Sample based • Complex valued • *M*-by-*N* matrix • Each output column contains the *M*-point DFT of the corresponding input channel in linear or bit-reversed order.

Valid Block Inputs • Real- or complex-valued • Must be in linear order • M must be a power of two.	Dimension Along Which Block Computes DFT	Corresponding Block Output Characteristics Output port rate = input port rate
Sample-based 1-by-*M* *row vector*	Row	• Sample based • Complex valued • 1-by-*M* row vector • Each output row contains the *M*-point DFT of the corresponding input channel in linear or bit-reversed order.
Unoriented length-*M* 1-D vector	Vector	Unoriented, length-*M*, complex-valued 1-D output vector containing *M*-point DFT of input in linear or bit-reversed order

Selecting the Twiddle Factor Computation Method

The **Twiddle factor computation** parameter determines how the block computes the necessary sine and cosine terms to calculate the term $e^{-j2\pi(m-1)(k-1)/M}$, shown in the first equation of this block reference page. This parameter has two settings, each with its advantages and disadvantages, as described in the following table. Note that only Table lookup mode is supported for fixed-point signals.

Twiddle Factor Computation Parameter Setting	Sine and Cosine Computation Method	Effect on Block Performance
Table lookup	The block computes and stores the trigonometric values before the simulation starts, and retrieves them during the simulation. When you generate code from the block, the processor running the generated code stores the trigonometric values computed by the block, and retrieves the values during code execution.	The block usually runs much more quickly, but requires extra memory for storing the precomputed trigonometric values. You can optimize the table for memory consumption or speed, as described in "Optimizing the Table of Trigonometric Values" on page 10-404.
Trigonometric fcn	The block computes sine and cosine values during the simulation. When you generate code from the block, the processor running the generated code computes the sine and cosine values while the code runs.	The block usually runs more slowly, but does not need extra data memory. For code generation, the block requires a support library to emulate the trigonometric functions, increasing the size of the generated code.

Optimizing the Table of Trigonometric Values

When you set the **Twiddle factor computation** parameter to Table lookup, you need to also set the **Optimize table for** parameter. This parameter optimizes the table of trigonometric values for speed or memory by varying the number of table entries as summarized in the following table.

Optimize Table for Parameter Setting	Number of Table Entries for N-Point FFT	Memory Required for Single-Precision 512-Point FFT
Speed	$3N/4$ — floating point N — fixed point	$\left(\dfrac{3 \times 512}{4}\text{table entries}\right) \times \left(4\dfrac{\text{bytes}}{\text{table entry}}\right)$ $= 1536$ bytes
Memory	$N/4$ — floating point Not supported for fixed point	$\left(\dfrac{512}{4}\text{table entries}\right) \times \left(4\dfrac{\text{bytes}}{\text{table entry}}\right)$ $= 512$ bytes

Ordering Output Column Entries

You can set the **Output in bit-reversed order** parameter to specify the ordering of the column elements of the output as either linear or bit-reversed order. If you select the **Output in bit-reversed order** check box, the block's output is in bit-reversed order. If you clear the **Output in bit-reversed order** check box, the block's output is in linear order.

Note With the FFT block, linearly ordering the output requires a butterfly operation. So, it might be better to output in bit-reversed order in some situations.

For more information ordering of the output, see "Linear and Bit-Reversed Output Order" on page 4-18.

Algorithms Used for FFT Computation

Depending on whether the block's input is floating-point or fixed-point, real- or complex-valued, and whether you want the output in linear or bit-reversed order, the block uses one or more of the following algorithms as summarized in the following tables:

- Butterfly operation
- Double-signal algorithm

- Half-length algorithm

- Radix-2 decimation-in-time (DIT) algorithm

- Radix-2 decimation-in-frequency (DIF) algorithm

For Floating-Point Signals:

Complexity of Input	Output Ordering	Algorithms Used for FFT Computation
Complex	Linear	Butterfly operation and radix-2 DIT
Complex	Bit-reversed	Radix-2 DIF
Real	Linear	Butterfly operation and radix-2 DIT in conjunction with the half-length and double-signal algorithms
Real	Bit-reversed	Radix-2 DIF in conjunction with the half-length and double-signal algorithms

For Fixed-Point Signals:

Complexity of Input	Output Ordering	Algorithms Used for FFT Computation
Real or complex	Linear	Butterfly operation and radix-2 DIT
Real or complex	Bit-reversed	Radix-2 DIF

For more information on the double-signal and half-length algorithms, see Proakis, John G. and Dimitris G. Manolakis. *Digital Signal Processing*. 3rd ed. Upper Saddle River, NJ: Prentice Hall, 1996. The section entitled "Efficient Computation of the DFT of Two Real Sequences" on page 475 describes the double signal algorithm. The section "Efficient Computation of the DFT of a 2N-Point Real Sequence" on page 476 describes the half-length algorithm.

Fixed-Point Data Types

The diagrams below show the data types used within the FFT block for fixed-point signals. You can set the sine table, accumulator, product output, and output data types displayed in the diagrams in the FFT block dialog as discussed in "Dialog Box" on page 10-408.

Inputs to the FFT block are first cast to the output data type and stored in the output buffer. Each butterfly stage then processes signals in the accumulator data type, with the final output of the butterfly being cast back into the output data type. A twiddle factor is multiplied in before each butterfly stage in a decimation-in-time FFT, and after each butterfly stage in a decimation-in-frequency FFT.

Decimation-in-Time FFT
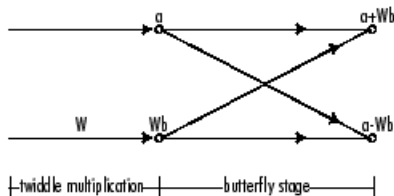
Decimation-in-Frequency FFT

Butterfly Stage Data Types

Twiddle Multiplication Data Types

The output of the multiplier is in the accumulator data type since both of the inputs to the multiplier are complex. For details on the complex multiplication performed, refer to "Multiplication Data Types" on page 8-16.

Dialog Box

The **Main** pane of the FFT block dialog appears as follows:

Twiddle factor computation

Specify the computation method of the term $e^{-j2\pi(m-1)(k-1)/M}$, shown in the first equation of this block reference page. In Table lookup mode, the block computes and stores the sine and

cosine values before the simulation starts. In `Trigonometric fcn` mode, the block computes the sine and cosine values during the simulation. See "Selecting the Twiddle Factor Computation Method" on page 10-403.

This parameter must be set to `Table lookup` for fixed-point signals.

Optimize table for

Select the optimization of the table of sine and cosine values for `Speed` or `Memory`. This parameter is only available when the **Twiddle factor computation** parameter is set to `Table lookup`. See "Selecting the Twiddle Factor Computation Method" on page 10-403.

This parameter must be set to `Speed` for fixed-point signals.

Output in bit-reversed order

Designate the order of the output channel elements relative to the ordering of the input elements. When selected, the output channel elements are in bit-reversed order relative to the input ordering. Otherwise, the output column elements are linearly ordered relative to the input ordering.

Linearly ordering the output requires extra data sorting manipulation, so in some situations it might be better to output in bit-reversed order.

The **Fixed-point** pane of the FFT block dialog appears as follows:
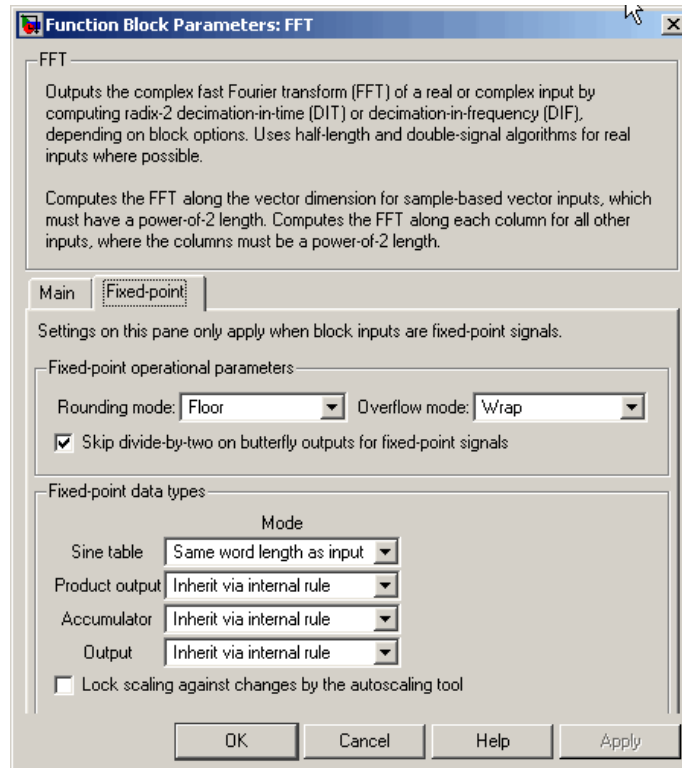
Rounding mode

    Select the rounding mode for fixed-point operations. The sine table values do not obey this parameter; they always round to Nearest.

Overflow mode

    Select the overflow mode for fixed-point operations. The sine table values do not obey this parameter; they are always saturated.

Skip divide-by-two on butterfly outputs for fixed-point signals

    When you select this parameter, no scaling occurs. When you do not select this parameter, the output of each butterfly of the FFT is divided by two for fixed-point signals.

Sine table

Choose how you will specify the word length of the values of the sine table. The fraction length of the sine table values is always equal to the word length minus one:

- When you select Same word length as input, the word length of the sine table values will match that of the input to the block.

- When you select Specify word length, you are able to enter the word length of the sine table values, in bits.

The sine table values do not obey the **Rounding mode** and **Overflow mode** parameters; they are always saturated and rounded to Nearest.

Product output

Use this parameter to specify how you would like to designate the product output word and fraction lengths. Refer to "Fixed-Point Data Types" on page 10-8 and "Multiplication Data Types" on page 8-16 for illustrations depicting the use of the product output data type in this block:

- When you select Inherit via internal rule, the product output word length and fraction length are automatically set according to the following equations:

$$product\ output\ word\ length\ =\\ output\ word\ length + sine\ table\ values\ word\ length$$

$$product\ output\ fraction\ length\ =\\ output\ fraction\ length + sine\ table\ values\ fraction\ length$$

- When you select Same as input, these characteristics will match those of the input to the block.

- When you select Binary point scaling, you are able to enter the word length and the fraction length of the product output, in bits.

- When you select Slope and bias scaling, you are able to enter the word length, in bits, and the slope of the product

output. This block requires power-of-two slope and a bias of zero.

Accumulator

Use this parameter to specify how you would like to designate the accumulator word and fraction lengths. Refer to "Fixed-Point Data Types" on page 10-8 and "Multiplication Data Types" on page 8-16 for illustrations depicting the use of the accumulator data type in this block:

- When you select Inherit via internal rule, the accumulator word length and fraction length are automatically set according to the following equations:

$$accumulator\ word\ length\ =\ product\ output\ word\ length + 1$$

$$accumulator\ fraction\ length\ =\ product\ output\ fraction\ length$$

- When you select Same as product output, these characteristics will match those of the product output.

- When you select Same as input, these characteristics will match those of the input to the block.

- When you select Binary point scaling, you are able to enter the word length and the fraction length of the accumulator, in bits.

- When you select Slope and bias scaling, you are able to enter the word length, in bits, and the slope of the accumulator. This block requires power-of-two slope and a bias of zero.

Output

Choose how you will specify the output word length and fraction length:

- When you select Inherit via internal rule, the output word length and fraction length are automatically set according to the following equations:

$$output\ word\ length\ =\ input\ word\ length$$
$$+\ floor(\log 2(FFT\ length - 1)) + 1$$

$$output\ fraction\ length\ =\ input\ fraction\ length$$

- When you select Same as input, these characteristics will match those of the input to the block.

- When you select Binary point scaling, you are able to enter the word length and the fraction length of the output, in bits.

- When you select Slope and bias scaling, you are able to enter the word length, in bits, and the slope of the output. This block requires power-of-two slope and a bias of zero.

References   Proakis, John G. and Dimitris G. Manolakis. *Digital Signal Processing.* 3rd ed. Upper Saddle River, NJ: Prentice Hall, 1996.

Supported Data Types
- Double-precision floating point

- Single-precision floating point

- Fixed point (signed only)

- 8-, 16-, and 32-bit signed integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

DCT	Signal Processing Blockset
IFFT	Signal Processing Blockset
Pad	Signal Processing Blockset
Zero Pad	Signal Processing Blockset
bitrevorder	Signal Processing Toolbox

FFT

fft	Signal Processing Toolbox
ifft	Signal Processing Toolbox

Purpose Construct filter realizations using the Digital Filter block or the Sum, Gain, and Delay blocks

Library Filtering / Filter Designs

Description
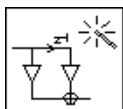
Note Use this block to implement fixed-point or floating-point digital filters using Sum, Gain, and Delay blocks or the Digital Filter block. You can either design a filter by using the block's filter design and analysis parameters, or import the coefficients of a filter you have designed elsewhere.

The following blocks also implement digital filters, but serve slightly different purposes:

• Digital Filter — Use to implement floating-point or fixed-point filters that you have already designed

• Digital Filter Design — Use to design, analyze, and then implement floating-point filters.

The Filter Realization Wizard is a tool for automatically implementing a digital filter. You must specify a filter, its structure, and the data types for the filter's inputs, outputs, and computations. The filter can support double-precision, single-precision, or fixed-point data types.

The Filter Realization Wizard can implement a digital filter in one of two ways. It can use a Digital Filter block, or it can create a subsystem block that implements the specified filter using Sum, Gain, and Delay blocks. If the Filter Realization Wizard creates a Digital Filter block, double-click the block to open the Block Parameters: Filter dialog box. If it creates a subsystem, double-click the subsystem block to see the filter implementation as shown in the figure below.

Filter Realization Wizard
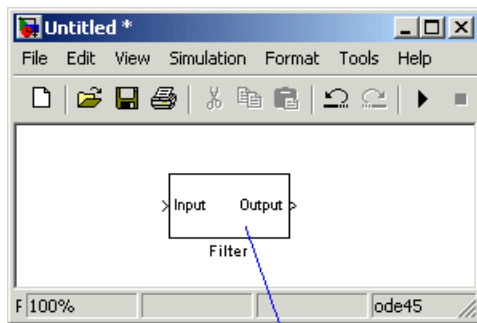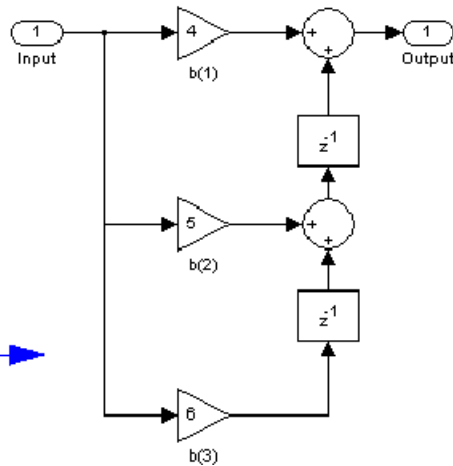
The subsystem block applies the specified filter to any sample-based input signal, or any frame-based row vector signal, and outputs the result. For more information about filter implementation, see "Specifying the Filter Implementation" on page 10-419.

The parameters of the Filter Realization Wizard are a part of a larger GUI, the Filter Design and Analysis Tool (FDATool), from the Signal Processing Toolbox. You can use the tools in FDATool to design and analyze your filter, and then use the Filter Realization Wizard parameters to implement the filter in your models.

Sections of This Reference Page

- "Valid Inputs and Corresponding Outputs" on page 10-417
- "Specifying the Filter and Its Data Type Support" on page 10-417
- "Supported Filter Structures" on page 10-419
- "Specifying the Filter Implementation" on page 10-419
- "Corresponding Method for dfilt" on page 10-421
- "Dialog Box" on page 10-422
- "References" on page 10-424

Valid Inputs and Corresponding Outputs

When the Filter Realization Wizard implements the specified filter by creating a new subsystem block, the block applies the specified filter to an input signal and outputs the result.

Valid Inputs

The subsystem block accepts inputs that are

- Sample-based vectors and matrices

- Frame-based row vectors (nonrecursive structures only)

Corresponding Outputs

The output of the subsystem block has the same dimensions and frame status as the input.

What Is Considered an Independent Channel

The subsystem block treats each *element* of a vector or matrix as an independent channel.

Specifying the Filter and Its Data Type Support

To specify a purely double-precision filter, you can either design a filter using the **Design Filter** panel, or import a filter using the **Import Filter** panel. (You can import dfilt filter objects as well as vectors of filter coefficients designed using functions in the Signal Processing Toolbox and the Filter Design Toolbox.)

You can also specify a fixed-point filter or a single-precision filter. You can specify such filters by using the **Set Quantization Parameters** panel, which requires the Filter Design Toolbox.

Filter Realization Wizard

> **Note** *Running* a model containing implementations of fixed-point filters requires "Simulink Fixed Point", but you can still edit models containing such filter implementations without Simulink Fixed Point. See the Simulink Fixed Point documentation for more information.

See the following topics to learn how to use the panels to specify your filter:

- For more information on the **Design Filter** panel, see "Filter Design and Analysis Tool (FDATool)" in the Signal Processing Toolbox documentation.

- For more information on the **Import Filter** panel, see "Importing a Filter Design" in the Signal Processing Toolbox documentation.

- For more information on the **Set Quantization Parameters** panel, see "Switching FDATool to Quantization Mode" in the Filter Design Toolbox documentation.

To open a panel, click the appropriate button in the lower-left corner of FDATool.
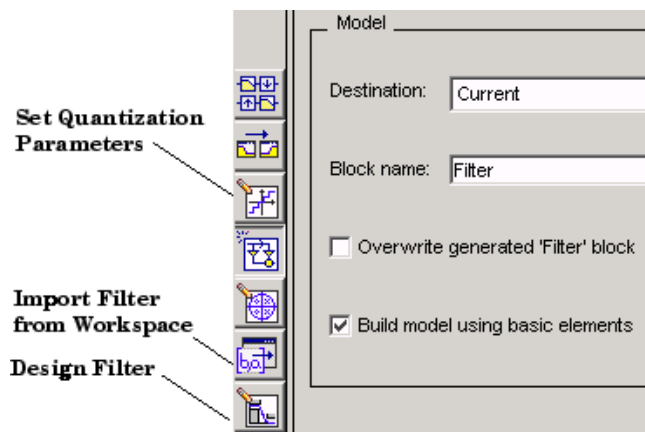
Supported Filter Structures

The Filter Realization Wizard supports the following structures:

- Direct form I
- Direct form II
- Direct form I transposed
- Direct form II transposed
- Second order sections for direct form I and II, and their transposes
- Direct form FIR
- Direct form FIR transposed
- Direct form antisymmetric FIR
- Direct form symmetric FIR
- Lattice ARMA
- Lattice AR
- Lattice MA (same as lattice minimum phase)
- Lattice all-pass
- Lattice maximum phase
- Cascade
- Parallel

Specifying the Filter Implementation

You can determine how the Filter Realization Wizard models the specified filter using the **Build model using basic elements** check box. When you select this check box, the Filter Realization Wizard creates a subsystem block that implements your filter using Sum, Gain, and Delay blocks. When you clear this check box, the Filter Realization Wizard uses a Digital Filter block to implement your filter. The **Build**

model using basic elements check box is only available when your filter can be implemented using a Digital Filter block.

If you have the Signal Processing Blockset, the Signal Processing Toolbox, and the Filter Design Toolbox installed on your system, the Filter Realization Wizard can generate a subsystem that represents either a double-precision or fixed-point filter. You must install "Simulink Fixed Point" to simulate a fixed-point filter. You can still edit the blocks used to implement the filter without installing Simulink Fixed Point.

Double-precision filter implemented with Sum, Gain, and Delay blocks
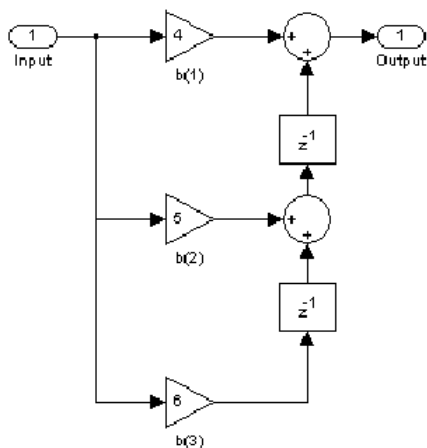
Fixed-point filter implemented with Sum, Gain, Delay, and Conversion blocks
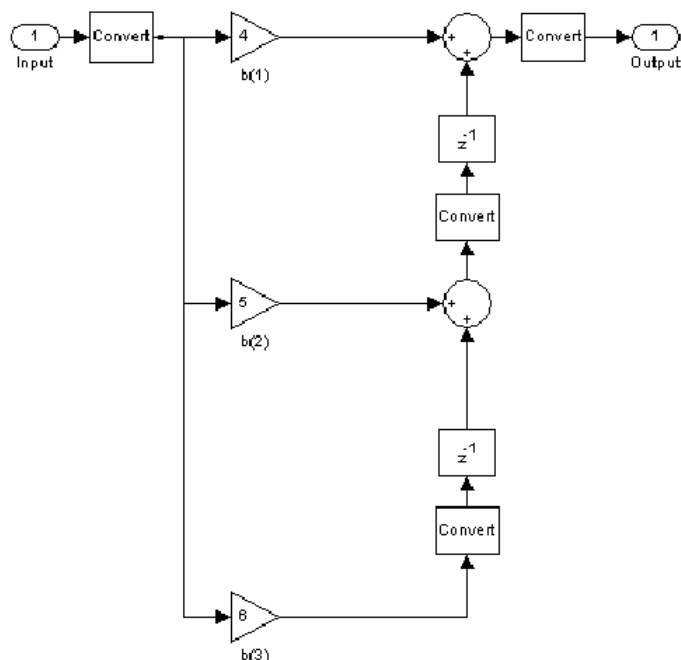
Implementations of Double-Precision and Fixed-Point Filters

Corresponding Method for dfilt

The dfilt (digital filter) object in the Signal Processing Toolbox has a method, realizemdl, that allows you to access the capabilities of the Filter Realization Wizard from the command line.

For more information about the realizemdl method, see the following:

- The topic on "Methods" in the dfilt reference page in the Signal Processing Toolbox documentation
- The realizemdl reference page in the Filter Design Toolbox documentation

Filter Realization Wizard

**Dialog
Box**

Note The following parameters for the Filter Realization Wizard are in the **Realize Model** pane of the Filter Design and Analysis Tool (FDATool) GUI. To open different panels of FDATool, click the different buttons at the lower-left corner. For more information about relevant panels, see "Specifying the Filter and Its Data Type Support" on page 10-417.
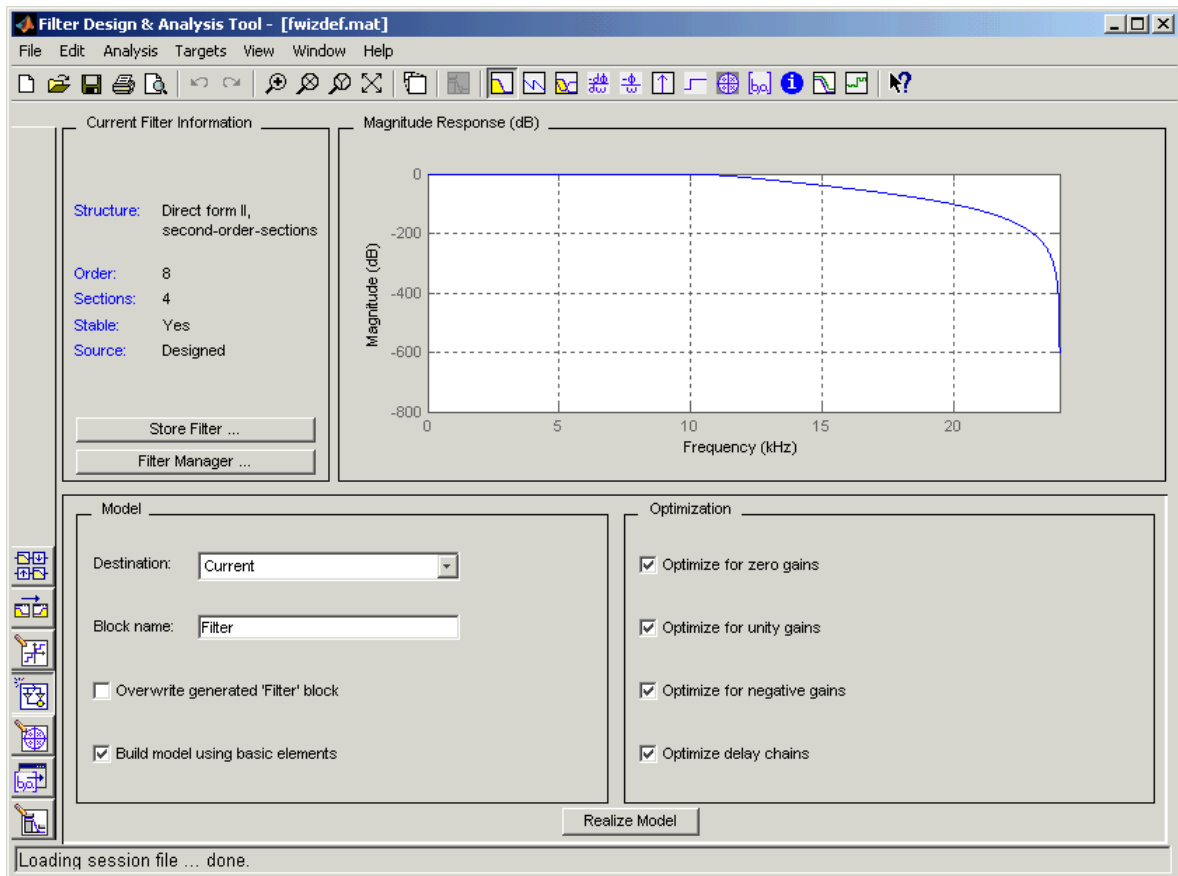
Destination

Specify where the new filter block should be created. This can be in a new model or in the current (most recently selected) model.

Block Name

Enter the name of the new filter block.

Overwrite generated block "Filter" block

When selected, the block overwrites any filter block in the current model with the name specified in the **Block Name** parameter. This parameter is enabled when the **Destination** parameter is set to Current.

Build model using basic elements

Select this check box to implement your filter using Sum, Gain, and Delay blocks. Clear this check box to implement your filter using the Digital Filter block. This parameter is only available when your filter can be modeled using the Digital Filter block.

Note that when your filter is implemented using Sum, Gain, and Delay blocks, inputs to the filter must be sample based.

Optimize for zero gains

When selected, the block removes zero-gain paths from the filter structure. For an example, see "Optimizing the Filter Structure" on page 3-49.

Optimize for unity gains

When selected, the block substitutes gains equal to 1 with a wire (short circuit). For an example, see "Optimizing the Filter Structure" on page 3-49.

Optimize for negative gains

When selected, the block substitutes gains equal to -1 with a wire (short circuit), and changes the corresponding sums to subtractions. For an example, see "Optimizing the Filter Structure" on page 3-49.

Filter Realization Wizard

Optimize delay chains

When selected, the block substitutes any delay chains made up of n unit delays with a single delay by n. For an example, see "Optimizing the Filter Structure" on page 3-49.

Realize Model

Click to create a subsystem block that implements the specified filter using Sum, Gain, and Delay blocks. To see the filter implementation, double-click the subsystem block. The subsystem block applies the specified filter to any sample-based input signal or frame-based row vector signal, and outputs the result.

Note For more information about relevant parameters in other panels of FDATool, see "Specifying the Filter and Its Data Type Support" on page 10-417.

References   Oppenheim, A. V. and R. W. Schafer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989.

Proakis, J. and D. Manolakis. *Digital Signal Processing*. 3rd ed. Englewood Cliffs, NJ: Prentice-Hall, 1996.

Supported Data Types

- Double-precision floating point

- Single-precision floating point — Supported only when you install the Filter Design Toolbox and "Simulink Fixed Point"

- Fixed point (signed and unsigned) — Supported only when you install the Filter Design Toolbox, Simulink Fixed Point, and the Fixed-Point Toolbox

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Digital Filter	Signal Processing Blockset
Digital Filter Design	Signal Processing Blockset
`filter`	Filter Design Toolbox
`realizemdl`	Filter Design Toolbox
`dfilt`	Signal Processing Toolbox
`filter`	Signal Processing Toolbox

- Chapter 3, "Filters" — Examples of when and how to use Signal Processing Blockset filtering blocks

- "Choosing Between Filter Design Blocks" on page 3-20

FIR Decimation

Purpose   Filter and downsample input signals

Library   Filtering / Multirate Filters

Description

x[2n]
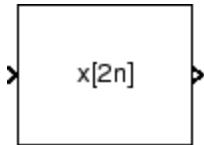
The FIR Decimation block resamples the discrete-time input at a rate K times slower than the input sample rate, where the integer K is specified by the **Decimation factor** parameter. This process consists of two steps:

- The block filters the input data using a direct-form FIR filter.

- The block downsamples the filtered data to a lower rate by discarding K-1 consecutive samples following every sample retained.

The FIR Decimation block implements the above FIR filtering and downsampling steps together using a polyphase filter structure, which is more efficient than straightforward filter-then-decimate algorithms. See N.J. Fliege, *Multirate Digital Signal Processing: Multirate Systems, Filter Banks, Wavelets* for more information.

The **FIR filter coefficients** parameter specifies the numerator coefficients of the FIR filter transfer function H(z).

$$H(z) = B(z) = b_1 + b_2 z^{-1} + \ldots + b_m z^{-(m-1)}$$

The length-m coefficient vector, [b(1) b(2) ... b(m)], can be generated by one of the filter design functions in the Signal Processing Toolbox, such as the `fir1` function used in the example below. The filter should be lowpass with normalized cutoff frequency no greater than $1/K$. All filter states are internally initialized to zero.

The FIR Decimation block supports real and complex floating-point and fixed-point inputs. This block supports triggered subsystems when you select `Maintain input frame rate` for the **Framing** parameter.

Sample-Based Operation

An M-by-N sample-based matrix input is treated as $M*N$ independent channels, and the block decimates each channel over time. The output

sample period is K times longer than the input sample period ($T_{so} = KT_{si}$), and the input and output sizes are identical.

Frame-Based Operation

An M_i-by-N frame-based matrix input is treated as N independent channels, and the block decimates each channel over time. The **Framing** parameter determines how the block adjusts the rate at the output to accommodate the reduced number of samples. There are two available options:

- `Maintain input frame size`

  The block generates the output at the decimated rate by using a proportionally longer frame *period* at the output port than at the input port. For decimation by a factor of K, the output frame period is K times longer than the input frame period ($T_{fo} = KT_{fi}$), but the input and output frame sizes are equal.

  The example below shows a single-channel input with a frame period of 1 second (**Sample time** = `1/64` and **Samples per frame** = `64` in the Signal From Workspace block) being decimated by a factor of 4 to a frame period of 4 seconds. The input and output frame sizes are identical.
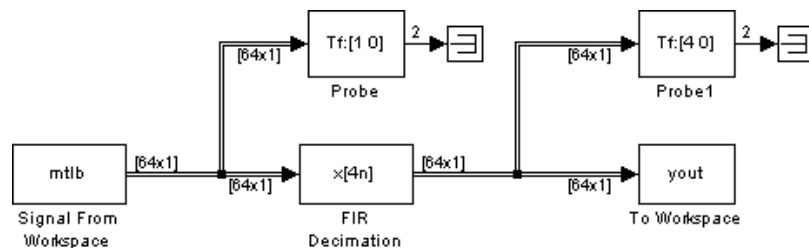
- `Maintain input frame rate`

  The block generates the output at the decimated rate by using a proportionally smaller frame *size* than the input. For decimation by a factor of K, the output frame size is K times smaller than the input frame size ($M_o = M_i/K$), but the input and output frame rates are

equal. The input frame size, M_i, must be a multiple of the decimation factor, K.

The example below shows a single-channel input of frame size 64 being decimated by a factor of 4 to a frame size of 16. The block's input and output frame rates are identical.
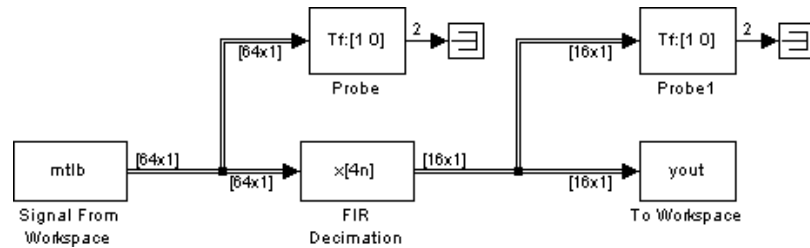
Latency

Zero Latency

The FIR Decimation block has *zero tasking latency* for all single-rate operations. The block is single-rate for the particular combinations of sampling mode and parameter settings shown in the table below.

Sampling Mode	Parameter Settings
Sample based	**Decimation factor** parameter, K, is 1.
Frame based	**Decimation factor** parameter, K, is 1, *or* **Framing** parameter is `Maintain input frame rate`.

Note that in sample-based mode, single-rate operation occurs only in the trivial case of factor-of-1 decimation.

The block also has zero latency for sample-based multirate operations in the Simulink single-tasking mode. Zero tasking latency means that

the block propagates the first filtered input sample (received at t=0) as the first output sample, followed by filtered input samples K+1, $2K$+1, and so on.

Nonzero Latency

The FIR Decimation block is multirate for all settings other than those in the previous table. The amount of latency for multirate operation depends on the Simulink tasking mode and the block's sampling mode, as shown in the following table.

Multirate...	Sample-Based Latency	Frame-Based Latency
Single-tasking	None	One frame (M_i samples)
Multitasking	One sample	One frame (M_i samples)

In cases of *one-sample latency*, a zero initial condition appears as the first output sample in each channel. The first filtered input sample appears as the second output sample, followed by filtered input samples K+1, $2K$+1, and so on.

In cases of *one-frame latency*, the first M_i output rows contain zeros, where M_i is the input frame size. The first filtered input sample (first filtered row of the input matrix) appears in the output as sample M_i+1, followed by filtered input samples K+1, $2K$+1, and so on. See the following example for an illustration of this case.

When the block exhibits latency, enter a value in the **Output buffer initial conditions** text box to specify the value to output at the output port until the first filtered input sample is available. The default initial condition value is 0.

FIR Decimation

Fixed-Point Data Types

The following diagram shows the data types used within the FIR Decimation block for fixed-point signals.
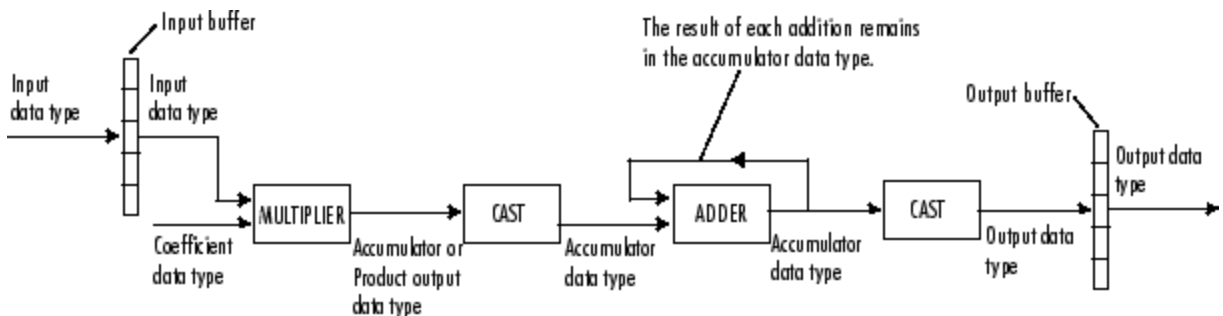
You can set the coefficient, product output, accumulator, and output data types in the block dialog as discussed in "Dialog Box" on page 10-434. The diagram shows that input data is stored in the input buffer in the same data type and scaling as the input. Filtered data is stored in the output buffer in the output data type and scaling that you set in the block dialog. Any initial conditions are also stored in the output buffer in the output data type and scaling you set in the block dialog.

The output of the multiplier is in the product output data type when at least one of the inputs to the multiplier is real. When both of the inputs to the multiplier are complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed, see "Multiplication Data Types" on page 8-16.

Examples   **Example 1**

Construct the frame-based model shown below.

Adjust the block parameters as follows:

- Configure the Signal From Workspace block to generate a two-channel signal with frame size of 4 and sample period of 0.25. This represents an output frame period of 1 (0.25*4). The first channel should contain the positive ramp signal 1, 2, ..., 100, and the second channel should contain the negative ramp signal -1, -2, ..., -100.

  - **Signal** = `[(1:100)' (-1:-1:-100)']`

  - **Sample time** = `0.25`

  - **Samples per frame** = `4`

- Configure the FIR Decimation block to decimate the two-channel input by decreasing the output frame rate by a factor of 2 relative to the input frame rate. Use a third-order filter with normalized cutoff frequency, f_{n0}, of 0.25. (Note that f_{n0} satisfies $f_{n0} \leq 1/K$.)

  - **FIR filter coefficients =** `fir1(3,0.25)`

  - **Downsample factor** = `2`

  - **Framing** = `Maintain input frame size`

The filter coefficient vector generated by `fir1(3,0.25)` is

```
[0.0386 0.4614 0.4614 0.0386]
```

or, equivalently,

$$H(z) = B(z) = 0.0386 + 0.04614z^{-1} + 0.04614z^{-2} + 0.0386z^{-3}$$

- Configure the Probe blocks by clearing the **Probe width**, **Probe complex signal**, and **Probe signal dimensions** check boxes (if desired).

This model is multirate because there are at least two distinct sample rates, as shown by the two Probe blocks. To run this model in the Simulink multitasking mode, open the Configuration Parameters dialog box. From the list on the left side of the dialog box, click **Solver**. From the **Type** list, select `Fixed-step`, and from the **Solver** list, select `discrete (no continuous states)`. From the **Tasking mode for periodic sample times** list, select `MultiTasking`. Also set the **Stop time** to 30.

Run the model and look at the output, yout. The first few samples of each channel are shown below.

```
yout =
         0          0
         0          0
         0          0
         0          0
    0.0386    -0.0386
    1.5000    -1.5000
    3.5000    -3.5000
    5.5000    -5.5000
    7.5000    -7.5000
    9.5000    -9.5000
   11.5000   -11.5000
```

Since this is a frame-based multirate model, the first four (M_i) output rows are zero. The first filtered input matrix row appears in the output as sample 5 (that is, sample M_i+1).

Example 2

The Polyphase FIR Decimation demo (`polyphaseDec_demo`) illustrates the underlying polyphase implementations of the FIR Decimation block. Run the demo and view the results on the scope. The output of the FIR Decimation block is the same as the output of the Polyphase Decimation Filter block.

Example 3

The `dspmrf_menu` demo illustrates the use of the FIR Decimation block in a number of multistage multirate filters.

FIR Decimation

Dialog Box

The **Main** pane of the FIR Decimation block dialog appears as follows:
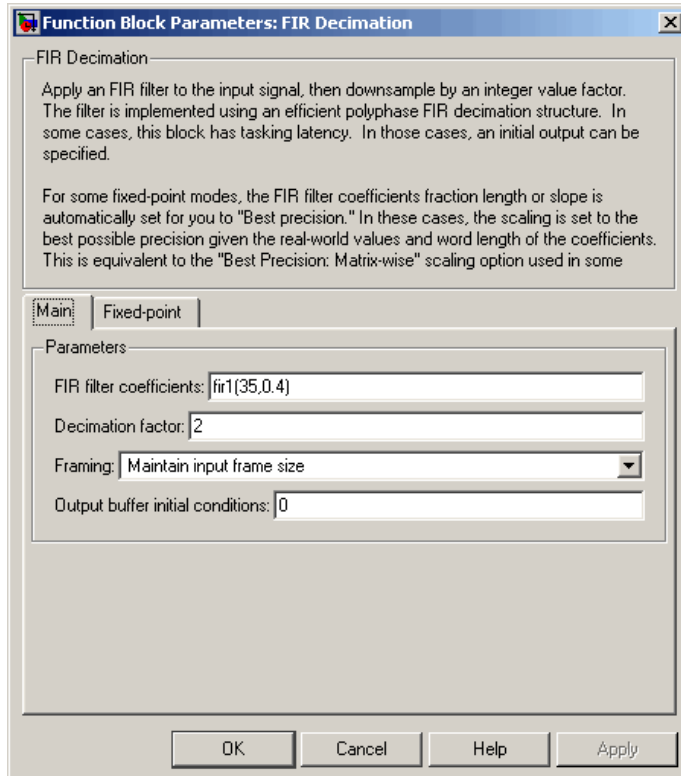
FIR filter coefficients

Specify the lowpass FIR filter coefficients, in descending powers of z.

Decimation factor

Specify the integer factor, K, by which to decrease the sample rate of the input sequence.

Framing

> For frame-based operation, specify the method by which to implement the decimation; reduce the output frame rate, or reduce the output frame size. This parameter can not be set to `Maintain input frame rate` for sample-based signals.

Output buffer initial conditions

> When the block exhibits latency, enter a value in the **Output buffer initial conditions** text box to specify the value to output at the output port until the first filtered input sample is available. The default initial condition value is zero.

The **Fixed-point** pane of the FIR Decimation block dialog appears as follows:

Rounding mode

Select the rounding mode for fixed-point operations. The filter coefficients do not obey this parameter; they always round to Nearest.

Overflow mode

Select the overflow mode for fixed-point operations. The filter coefficients do not obey this parameter; they are always saturated.

Coefficients

Choose how you will specify the word length and the fraction length of the filter coefficients:

- When you select Same word length as input, the word length of the filter coefficients will match that of the input to the block. In this mode, the fraction length of the coefficients is automatically set to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.

- When you select Specify word length, you are able to enter the word length of the coefficients, in bits. In this mode, the fraction length of the coefficients is automatically set to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.

- When you select Binary point scaling, you are able to enter the word length and the fraction length of the coefficients, in bits.

- When you select Slope and bias scaling, you are able to enter the word length, in bits, and the slope of the coefficients. This block requires power-of-two slope and a bias of zero.

The filter coefficients do not obey the **Rounding mode** and the **Overflow mode** parameters; they are always saturated and rounded to Nearest.

Product output

Use this parameter to specify how you would like to designate the product output word and fraction lengths. Refer to "Fixed-Point Data Types" on page 10-430 and "Multiplication Data Types" on page 8-16 for illustrations depicting the use of the product output data type in this block:

- When you select Inherit via internal rule, the product output word length and fraction length are automatically set according to the following equations:

$$\text{ideal product output word length } =\\ \text{input word length} + FIR \text{ coefficients word length}$$

$$\text{ideal product output fraction length } =\\ \text{input fraction length} + FIR \text{ coefficients fraction length}$$

Note The actual product output word length may be equal to or greater than the calculated ideal product output word length, depending on the settings on the **Hardware Implementation** pane of the Configuration Parameters dialog box.

- When you select Same as input, these characteristics will match those of the input to the block.

- When you select Binary point scaling, you are able to enter the word length and the fraction length of the product output, in bits.

- When you select Slope and bias scaling, you are able to enter the word length, in bits, and the slope of the product output. This block requires power-of-two slope and a bias of zero.

Accumulator

The result of each addition remains in the accumulator data type.

As depicted above, inputs to the accumulator are cast to the accumulator data type. The output of the adder remains in the accumulator data type as each element of the input is added to it. Use this parameter to specify how you would like to designate this accumulator word and fraction lengths.

You also use this parameter to specify the accumulator word and fraction lengths resulting from a complex-complex multiplication in the block. Refer to "Multiplication Data Types" on page 8-16 for more information.

- When you select Inherit via internal rule, the accumulator word length and fraction length are automatically set according to the following equations:

$$ideal\ accumulator\ word\ length\ =\ ideal\ product\ output\ word\ length \\ +\ \text{floor}(\log 2(number\ of\ accumulations)) + 1$$

$$ideal\ accumulator\ fraction\ length\ =\ ideal\ product\ output\ fraction\ length$$

where the number of accumulations is given by

$$((number\ of\ coefficients\ /\ decimation\ factor) - 1)$$

if either the coefficients or inputs are real

$$number\ of\ coefficients\ /\ decimation\ factor$$

if both the coefficients and inputs are complex

> **Note** The actual accumulator word length may be equal to or
> greater than the calculated ideal product output word length,
> depending on the settings on the **Hardware Implementation**
> pane of the Configuration Parameters dialog box.

- When you select `Same as product output`, these
  characteristics will match those of the product output.

- When you select `Same as input`, these characteristics will
  match those of the input to the block.

- When you select `Binary point scaling`, you are able to enter
  the word length and the fraction length of the accumulator,
  in bits.

- When you select `Slope and bias scaling`, you are able to
  enter the word length, in bits, and the slope of the accumulator.
  This block requires power-of-two slope and a bias of zero.

Output

Choose how you will specify the output word length and fraction
length:

- When you select `Same as accumulator`, these characteristics
  will match those of the accumulator.

  A special case occurs when `Inherit via internal rule` is
  specified for **Accumulator**, and block inputs and coefficients
  are complex. In that case, the output word length will be one
  less than the accumulator word length.

- When you select `Same as product output`, these
  characteristics will match those of the product output.

- When you select `Same as input`, these characteristics will
  match those of the input to the block.

- When you select `Binary point scaling`, you are able to enter
  the word length and the fraction length of the output, in bits.

FIR Decimation

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the output. This block requires power-of-two slope and a bias of zero.

Lock scaling against changes by the autoscaling tool
Select this parameter to prevent any fixed-point scaling you specify in this block mask from being overriden by the autoscaling tool in the Fixed-Point Settings interface. For more information about the autoscaling tool, refer to "Fixed-Point Settings Interface" on page 8-28.

References Fliege, N. J. *Multirate Digital Signal Processing: Multirate Systems, Filter Banks, Wavelets*. West Sussex, England: John Wiley & Sons, 1994.

Supported Data Types
- Double-precision floating point
- Single-precision floating point
- Fixed point (signed only)
- 8-, 16-, and 32-bit signed integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Downsample	Signal Processing Blockset
FIR Interpolation	Signal Processing Blockset
FIR Rate Conversion	Signal Processing Blockset
decimate	Signal Processing Toolbox
fir1	Signal Processing Toolbox

`fir2`	Signal Processing Toolbox
`firls`	Signal Processing Toolbox

FIR Interpolation

Purpose   Upsample and filter input signals

Library   Filtering / Multirate Filters
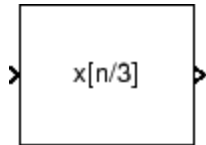
Description   The FIR Interpolation block resamples the discrete-time input at a
                     rate *L* times faster than the input sample rate, where the integer *L* is
                     specified by the **Interpolation factor** parameter. This process consists
                     of two steps:

x[n/3]

- The block upsamples the input to a higher rate by inserting *L*-1 zeros
  between samples.

- The block filters the upsampled data with a direct-form FIR filter.

The FIR Interpolation block implements the above upsampling and
FIR filtering steps together using a polyphase filter structure, which is
more efficient than straightforward upsample-then-filter algorithms.
See N.J. Fliege, *Multirate Digital Signal Processing: Multirate Systems,
Filter Banks, Wavelets* for more information.

The **FIR filter coefficients** parameter specifies the numerator
coefficients of the FIR filter transfer function H(*z*).

$$H(z) = B(z) = b_1 + b_2 z^{-1} + \dots + b_m z^{-(m-1)}$$

The coefficient vector, `[b(1) b(2) ... b(m)]`, can be generated by
one of the filter design functions in the Signal Processing Toolbox (such
as `fir1`), and should have a length greater than the interpolation factor
(*m*>L). The filter should be lowpass with normalized cutoff frequency no
greater than 1/*L*. All filter states are internally initialized to zero.

The FIR Interpolation block supports real and complex floating-point
and fixed-point inputs. This block supports triggered subsystems when
you select `Maintain input frame rate` for the **Framing** parameter.

Sample-Based Operation

An *M*-by-*N* sample-based matrix input is treated as *M**N independent
channels, and the block interpolates each channel over time. The output

sample period is L times shorter than the input sample period ($T_{so} = T_{si}/L$), and the input and output sizes are identical.

Frame-Based Operation

An M_i-by-N frame-based matrix input is treated as N independent channels, and the block decimates each channel over time. The **Framing** parameter determines how the block adjusts the rate at the output to accommodate the added samples. There are two available options:

- `Maintain input frame size`

  The block generates the output at the interpolated rate by using a proportionally shorter frame *period* at the output port than at the input port. For interpolation by a factor of L, the output frame period is L times shorter than the input frame period ($T_{fo} = T_{fi}/L$), but the input and output frame sizes are equal.

  The example below shows a single-channel input with a frame period of 1 second (**Sample time** = `1/64` and **Samples per frame** = `64` in the Signal From Workspace block) being interpolated by a factor of 4 to a frame period of 0.25 second. The input and output frame sizes are identical.
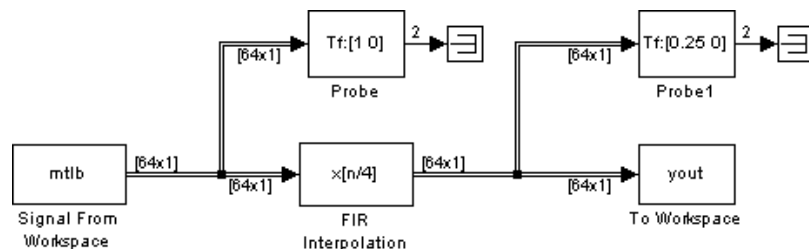
- `Maintain input frame rate`

  The block generates the output at the interpolated rate by using a proportionally larger frame *size* than the input. For interpolation by a factor of L, the output frame size is L times larger than the

input frame size ($M_o = M_i*L$), but the input and output frame rates are equal.

The example below shows a single-channel input of frame size 16 being interpolated by a factor of 4 to a frame size of 64. The block's input and output frame rates are identical.
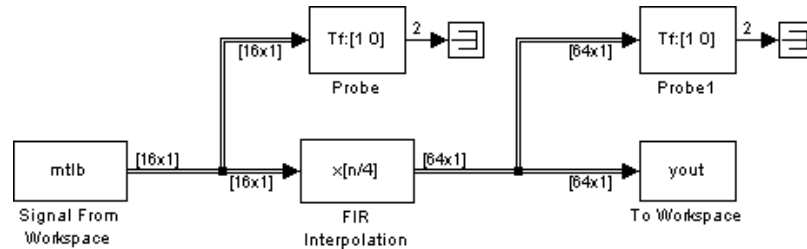
Latency

Zero Latency

The FIR Interpolation block has *zero tasking latency* for all single-rate operations. The block is single rate for the particular combinations of sampling mode and parameter settings shown in the table below.

Sampling Mode	Parameter Settings
Sample based	**Interpolation factor** parameter, *L*, is 1.
Frame based	**Interpolation factor** parameter, *L*, is 1, *or* **Framing** parameter is Maintain input frame rate.

Note that in sample-based mode, single-rate operation occurs only in the trivial case of factor-of-1 interpolation.

The block also has zero latency for sample-based multirate operations in the Simulink single-tasking mode. Zero tasking latency means that

the block propagates the first filtered input (received at t=0) as the first input sample, followed by L-1 interpolated values, the second filtered input sample, and so on.

Nonzero Latency

The FIR Interpolation block is multirate for all settings other than those in the previous table. The amount of latency for multirate operation depends on the Simulink tasking mode and the block's sampling mode, as shown in the following table.

Multirate...	Sample-Based Latency	Frame-Based Latency
Single-tasking	None	None
Multitasking	L samples	L frames (M_i samples per frame)

When the block exhibits latency, the default initial condition is zero. Alternatively, you can enter a value in the **Output buffer initial conditions** text box. This value is divided by the **Interpolation factor** and output at the output port until the first filtered input sample is available.

In sample-based cases, the scaled initial conditions appear at the start of each channel, followed immediately by the first filtered input sample, L-1 interpolated values, and so on.

In frame-based cases, with the default initial condition, the first M_iL output rows contain zeros, where M_i is the input frame size. The first filtered input sample (first filtered row of the input matrix) appears in the output as sample M_iL+1, followed by L-1 interpolated values, the second filtered input sample, and so on. See the following example for an illustration of this case.

FIR Interpolation

Fixed-Point Data Types

The following diagram shows the data types used within the FIR Interpolation block for fixed-point signals.
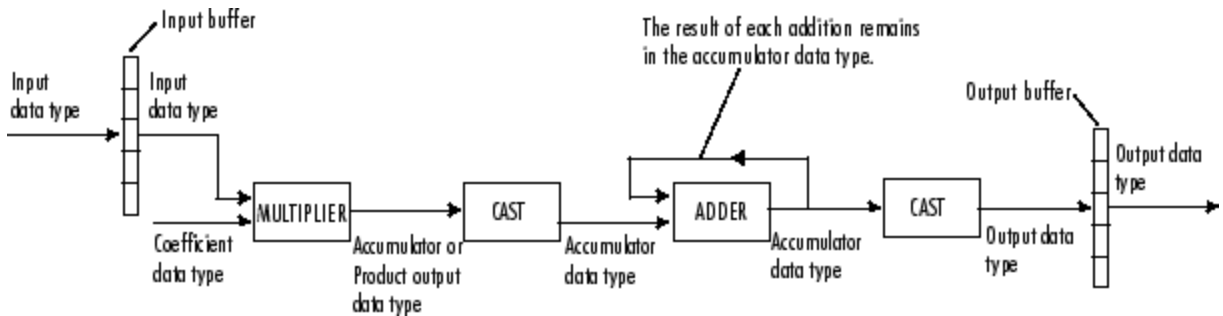
You can set the coefficient, product output, accumulator, and output data types in the block dialog as discussed in "Dialog Box" on page 10-450. The diagram shows that input data is stored in the input buffer in the same data type and scaling as the input. Filtered data is stored in the output buffer in the output data type and scaling that you set in the block dialog. Any initial conditions are also stored in the output buffer in the output data type and scaling you set in the block dialog.

The output of the multiplier is in the product output data type when at least one of the inputs to the multiplier is real. When both of the inputs to the multiplier are complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed, see "Multiplication Data Types" on page 8-16.

Examples

Example 1

Construct the frame-based model shown below.

Adjust the block parameters as follows:

- Configure the Signal From Workspace block to generate a two-channel signal with frame size of 4 and sample period of 0.25. This represents an output frame period of 1 (0.25*4). The first channel should contain the positive ramp signal 1, 2, ..., 100, and the second channel should contain the negative ramp signal -1, -2, ..., -100.

  - **Signal** = [(1:100)' (-1:-1:-100)']

  - **Sample time** = 0.25

  - **Samples per frame** = 4

- Configure the FIR Interpolation block to interpolate the two-channel input by increasing the output frame rate by a factor of 2 relative to the input frame rate. Use a third-order filter ($m=3$) with normalized cutoff frequency, f_{n0}, of 0.25. (Note that f_{n0} and m satisfy $f_{n0} \leq 1/L$ and $m > L$.)

  - **FIR filter coefficients =** fir1(3,0.25)

  - **Interpolation factor** = 2

  - **Framing** = Maintain input frame size

The filter coefficient vector generated by fir1(3,0.25) is

```
[0.0386 0.4614 0.4614 0.0386]
```

or, equivalently,

$$H(z) = B(z) = 0.0386 + 0.04614z^{-1} + 0.04614z^{-2} + 0.0386z^{-3}$$

- Configure the Probe blocks by clearing the **Probe width**, **Probe complex signal**, and **Probe signal dimensions** check boxes (if desired).

This model is multirate because there are at least two distinct sample rates, as shown by the two Probe blocks. To run this model in the Simulink multitasking mode, open the Configuration Parameters dialog box. In the **Select** pane, click **Solver**. From the **Type** list, select Fixed-step, and from the **Solver** list, select discrete (no continuous states). From the **Tasking mode for periodic sample times** list, select MultiTasking. Also set the **Stop time** to 30.

Run the model and look at the output, yout. The first few samples of each channel are shown below.

```
dsp_examples_yout =
        0          0
        0          0
        0          0
        0          0
        0          0
        0          0
        0          0
        0          0
   0.0386    -0.0386
   0.4614    -0.4614
   0.5386    -0.5386
   0.9614    -0.9614
   1.0386    -1.0386
```

Since we ran this frame-based multirate model in multitasking mode, the first eight (M_iL) output rows are zero. The first filtered input matrix row appears in the output as sample 9 (that is, sample M_iL+1). Every other row is an interpolated value.

Example 2

The Polyphase FIR Interpolation demo (`polyphaseInterp_demo`) illustrates the underlying polyphase implementations of the FIR Interpolation block. Run the demo and view the results on the scope. The output of the FIR Interpolation block is the same as the output of the Polyphase Interpolation Filter block.

Example 3

The `dspintrp` demo provides another simple example, and the `dspmrf_menu` demo illustrates the use of the FIR Interpolation block in a number of multistage multirate filters.

FIR Interpolation

Dialog Box

The **Main** pane of the FIR Interpolation block dialog appears as follows:
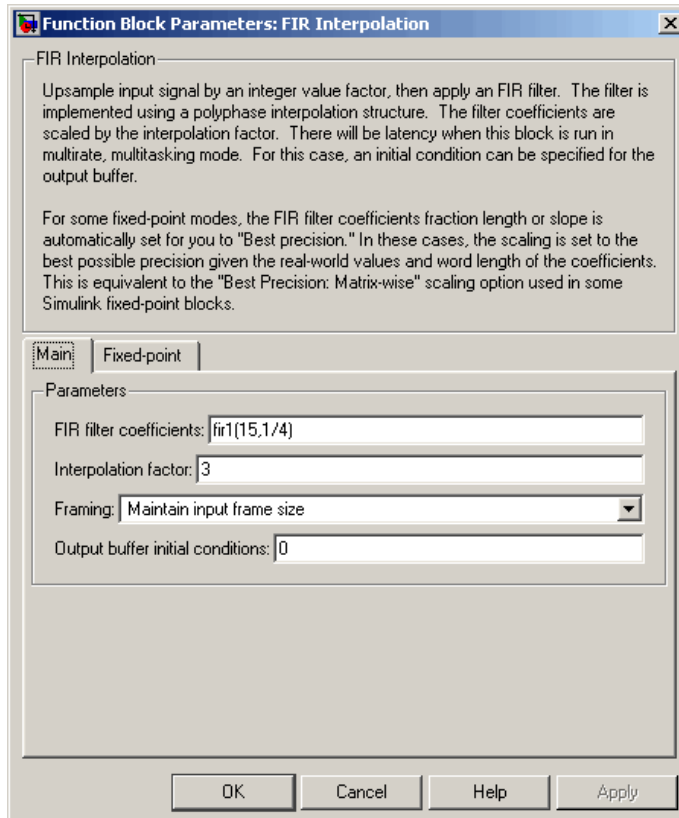
FIR filter coefficients

Specify the FIR filter coefficients, in descending powers of z.

Interpolation factor

Specify the integer factor, L, by which to increase the sample rate of the input sequence.

Framing

> For frame-based operation, specify the method by which to implement the interpolation: increase the output frame rate, or increase the output frame size. This parameter can not be set to `Maintain input frame rate` for sample-based signals.

Output buffer initial conditions

> When the block exhibits latency, enter a value in the **Output buffer initial conditions** text box to specify the value to output at the output port until the first filtered input sample is available. The default initial condition value is `0`.
>
> Output buffer initial conditions are stored in the output data type and scaling.

The **Fixed-point** pane of the FIR Interpolation block dialog appears as follows:

FIR Interpolation

Rounding mode

Select the rounding mode for fixed-point operations. The filter coefficients do not obey this parameter; they always round to Nearest.

Overflow mode

Select the overflow mode for fixed-point operations. The filter coefficients do not obey this parameter; they are always saturated.

Coefficients

Choose how you will specify the word length and fraction length of the filter coefficients:

- When you select `Same word length as input`, the word length of the filter coefficients will match that of the input to the block. In this mode, the fraction length of the coefficients is automatically set to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.

- When you select `Specify word length`, you are able to enter the word length of the coefficients, in bits. In this mode, the fraction length of the coefficients is automatically set to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the coefficients, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the coefficients. This block requires power-of-two slope and a bias of zero.

The filter coefficients do not obey the **Rounding mode** and the **Overflow mode** parameters; they are always saturated and rounded to `Nearest`.

Product output

Use this parameter to specify how you would like to designate the product output word and fraction lengths. Refer to "Fixed-Point Data Types" on page 10-446 and "Multiplication Data Types" on page 8-16 for illustrations depicting the use of the product output data type in this block:

- When you select `Inherit via internal rule`, the product output word length and fraction length are automatically set according to the following equations:

$$ideal\ product\ output\ word\ length\ =$$
$$input\ word\ length + FIR\ coefficients\ word\ length$$

$$ideal\ product\ output\ fraction\ length\ =$$
$$input\ fraction\ length + FIR\ coefficients\ fraction\ length$$

> **Note** The actual product output word length may be equal to or greater than the calculated ideal product output word length, depending on the settings on the **Hardware Implementation** pane of the Configuration Parameters dialog box.

- When you select `Same as input`, these characteristics will match those of the input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the product output, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the product output. This block requires power-of-two slope and a bias of zero.

Accumulator

As depicted above, inputs to the accumulator are cast to the accumulator data type. The output of the adder remains in the accumulator data type as each element of the input is added to

it. Use this parameter to specify how you would like to designate this accumulator word and fraction lengths.

You also use this parameter to specify the accumulator word and fraction lengths resulting from a complex-complex multiplication in the block. Refer to "Multiplication Data Types" on page 8-16 for more information:

- When you select Inherit via internal rule, the accumulator word length and fraction length are automatically set according to the following equations:

$$ideal\ accumulator\ word\ length = ideal\ product\ output\ word\ length \\ + \text{floor}(\log 2(number\ of\ accumulations)) + 1$$

$$ideal\ accumulator\ fraction\ length = \\ ideal\ product\ output\ fraction\ length$$

where the number of accumulations is given by

$$((number\ of\ coefficients / (interpolation\ factor)) - 1)$$
if either the coefficients or inputs are real
$$number\ of\ coefficients / (interpolation\ factor)$$
if both the coefficients and inputs are complex

> **Note** The actual accumulator word length may be equal to or greater than the calculated ideal product output word length, depending on the settings on the **Hardware Implementation** pane of the Configuration Parameters dialog box.

- When you select Same as product output, these characteristics will match those of the product output.

- When you select Same as input, these characteristics will match those of the input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the accumulator, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the accumulator. This block requires power-of-two slope and a bias of zero.

Output

Choose how you will specify the output word length and fraction length:

- When you select `Same as accumulator`, these characteristics will match those of the accumulator.

  A special case occurs when `Inherit via internal rule` is specified for **Accumulator**, and block inputs and coefficients are complex. In that case, the output word length will be one less than the accumulator word length.

- When you select `Same as product output`, these characteristics will match those of the product output.

- When you select `Same as input`, these characteristics will match those of the input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the output, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the output. This block requires power-of-two slope and a bias of zero.

Lock scaling against changes by the autoscaling tool

Select this parameter to prevent any fixed-point scaling you specify in this block mask from being overriden by the autoscaling tool in the Fixed-Point Settings interface. For more information about the autoscaling tool, refer to "Fixed-Point Settings Interface" on page 8-28.

References   Fliege, N. J. *Multirate Digital Signal Processing: Multirate Systems, Filter Banks, Wavelets*. West Sussex, England: John Wiley & Sons, 1994.

Supported Data Types
- Double-precision floating point
- Single-precision floating point
- Fixed point (signed only)
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

FIR Decimation	Signal Processing Blockset
FIR Rate Conversion	Signal Processing Blockset
Upsample	Signal Processing Blockset
fir1	Signal Processing Toolbox
fir2	Signal Processing Toolbox
firls	Signal Processing Toolbox
interp	Signal Processing Toolbox

FIR Rate Conversion

Purpose   Upsample, filter, and downsample input signals

Library   Filtering / Multirate Filters
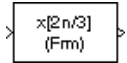
Description   The FIR Rate Conversion block resamples the discrete-time input
to a period K/L times the input sample period, where the integer K
is specified by the **Decimation factor** parameter and the integer L
is specified by the **Interpolation factor** parameter. The resampling
process consists of the following steps:

1 The block upsamples the input to a higher rate by inserting L-1 zeros
between input samples.

2 The upsampled data is passed through a direct-form II transpose
FIR filter.

3 The block downsamples the filtered data to a lower rate by discarding
K-1 consecutive samples following each sample retained.

K and L must be *relatively prime* integers; that is, the ratio K/L
cannot be reducible to a ratio of smaller integers. The FIR Rate
Conversion block implements the above three steps together using a
polyphase filter structure, which is more efficient than straightforward
upsample-filter-decimate algorithms. See N. J. Fliege [1] for more
information.

The **FIR filter coefficients** parameter specifies the numerator
coefficients of the FIR filter transfer function H(z).

$$H(z) = B(z) = b_1 + b_2 z^{-1} + \dots + b_m z^{-(m-1)}$$

The coefficient vector, `[b(1) b(2) ... b(m)]`, can be generated by
one of the filter design functions in the Signal Processing Toolbox (such
as `fir1`), and should have a length greater than the interpolation factor
($m>L$). The filter should be lowpass with normalized cutoff frequency
no greater than min(1/L,1/K). All filter states are internally initialized
to zero.

Frame-Based Operation

This block accepts *only* frame-based inputs. An M_i-by-N frame-based matrix input is treated as N independent channels, and the block resamples each channel independently over time.

The **Interpolation factor**, L, and **Decimation factor**, K, must satisfy the relation

$$\frac{K}{L} = \frac{M_i}{M_o}$$

for an *integer* output frame size M_o. The simplest way to satisfy this requirement is to let the **Decimation factor** equal the input frame size, M_i. The output frame size, M_o, is then equal to the **Interpolation factor**. This change in the frame size, from M_i to M_o, produces the desired rate conversion while leaving the output frame period the same as the input ($T_{fo} = T_{fi}$).
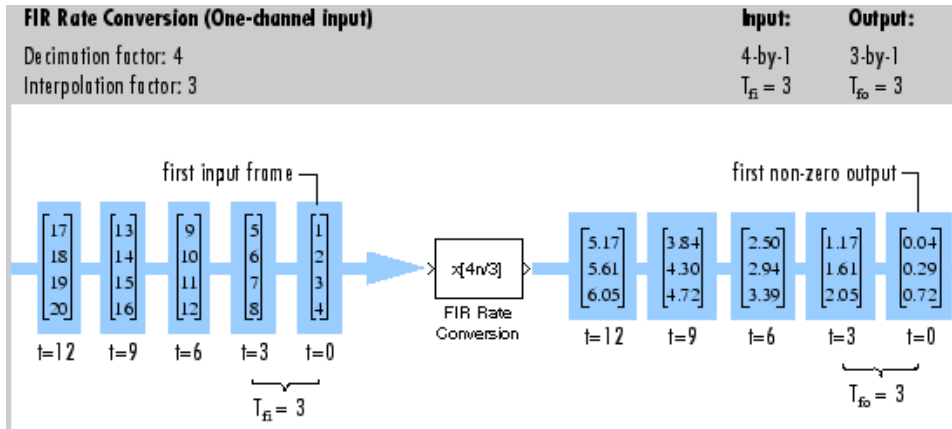
Latency

The FIR Rate Conversion block has no tasking latency. The block propagates the first filtered input (received at t=0) as the first output sample.

FIR Rate Conversion

Fixed-Point Data Types

The following diagram shows the data types used within the FIR Rate Conversion block for fixed-point signals.
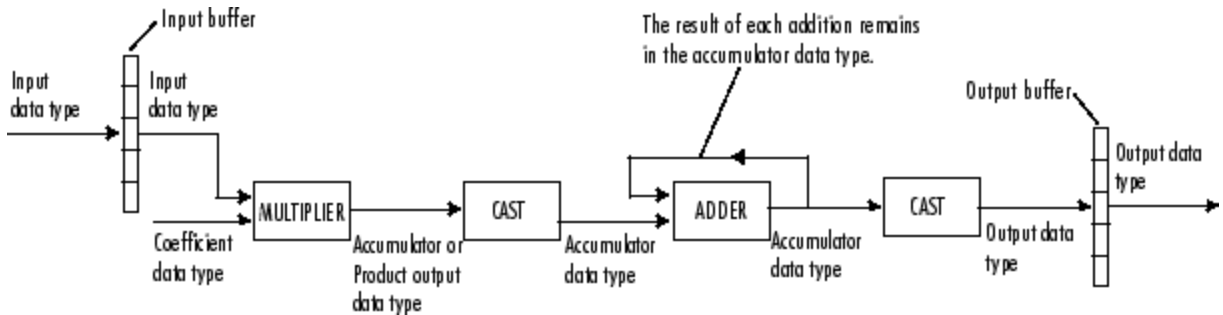
You can set the coefficient, product output, accumulator, and output data types in the block dialog as discussed in "Dialog Box" on page 10-462. The diagram shows that input data is stored in the input buffer in the same data type and scaling as the input. Filtered data is stored in the output buffer in the output data type and scaling that you set in the block dialog. Any initial conditions are also stored in the output buffer in the output data type and scaling you set in the block dialog.

The output of the multiplier is in the product output data type when at least one of the inputs to the multiplier is real. When both of the inputs to the multiplier are complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed, refer to "Multiplication Data Types" on page 8-16.

Examples   Example 1

The Rate Converter demo (`polyphaseUpFirDn_demo`) illustrates the underlying polyphase implementations of the FIR Rate Conversion block. Run the demo and view the results on the scope. The output of the FIR Rate Conversion block is the same as the output of the system comprised of the FIR Decimation block and FIR Interpolation block. The output of the FIR Rate Conversion block is also the same as the output of the Polyphase Filter block.

Diagnostics   An error is generated when the relation between K and L shown above is not satisfied.

```
(Input port width)/(Output port width) must equal the
(Decimation factor)/(Interpolation factor).
```

A warning is generated when L and K are not relatively prime; that is, when the ratio L/K can be reduced to a ratio of smaller integers.

```
Warning: Integer conversion factors are not relatively prime in
block 'modelname/FIR Rate Conversion (Frame)'. Converting ratio
L/M to l/m.
```

The block scales the ratio to be relatively prime and continues the simulation.

FIR Rate Conversion

Dialog Box

The **Main** pane of the FIR Rate Conversion block dialog appears as follows:
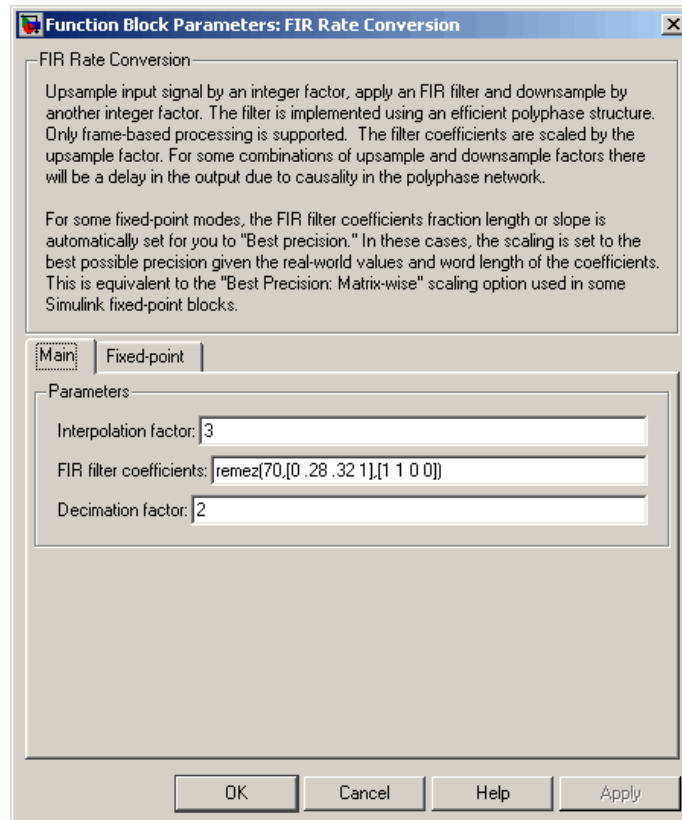
Interpolation factor

Specify the integer factor, L, by which to upsample the signal before filtering.

FIR filter coefficients

Specify the FIR filter coefficients in descending powers of z.

Decimation factor

Specify the integer factor, *K*, by which to downsample the signal after filtering.

The **Fixed-point** pane of the FIR Rate Conversion block dialog appears as follows:

Rounding mode

Select the rounding mode for fixed-point operations. The filter coefficients do not obey this parameter; they always round to Nearest.

Overflow mode

Select the overflow mode for fixed-point operations. The filter coefficients do not obey this parameter; they are always saturated.

Coefficients

Choose how you will specify the word length and fraction length of the filter coefficients.

- When you select `Same word length as input`, the word length of the filter coefficients will match that of the input to the block. In this mode, the fraction length of the coefficients is automatically set to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.

- When you select `Specify word length`, you are able to enter the word length of the coefficients, in bits. In this mode, the fraction length of the coefficients is automatically set to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the coefficients, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the coefficients. This block requires power-of-two slope and a bias of zero.

- The coefficients do not obey the **Round integer calculations toward** and the **Saturate on integer overflow** parameters; they are always saturated and rounded to `Nearest`.

Product output

Use this parameter to specify how you would like to designate the product output word and fraction lengths. Refer to "Fixed-Point Data Types" on page 10-430 and "Multiplication Data Types" on page 8-16 for illustrations depicting the use of the product output data type in this block.

- When you select `Inherit via internal rule`, the product output word length and fraction length are automatically set according to the following equations:

$$ideal\ product\ output\ word\ length = \\ input\ word\ length + FIR\ coefficients\ word\ length$$

$$ideal\ product\ output\ fraction\ length = \\ input\ fraction\ length + FIR\ coefficients\ fraction\ length$$

Note The actual product output word length may be equal to or greater than the calculated ideal product output word length, depending on the settings on the **Hardware Implementation** pane of the Configuration Parameters dialog box.

- When you select `Same as input`, these characteristics will match those of the input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the product output, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the product output. This block requires power-of-two slope and a bias of zero.

Accumulator

The result of each addition remains in the accumulator data type.

As depicted above, inputs to the accumulator are cast to the accumulator data type. The output of the adder remains in the accumulator data type as each element of the input is added to it. Use this parameter to specify how you would like to designate this accumulator word and fraction lengths.

You also use this parameter to specify the accumulator word and fraction lengths resulting from a complex-complex multiplication in the block. Refer to "Multiplication Data Types" on page 8-16 for more information.

- When you select Inherit via internal rule, the accumulator word length and fraction length are automatically set according to the following equations:

$$ideal\ accumulator\ word\ length\ =\ ideal\ product\ output\ word\ length \\ +\ \text{floor}(\log 2(number\ of\ accumulations)) + 1$$

$$ideal\ accumulator\ fraction\ length\ = \\ ideal\ product\ output\ fraction\ length$$

where the number of accumulations is given by

$$((number\ of\ coefficients / (interpolation\ factor)) - 1)$$

if either the coefficients or inputs are real

$$number\ of\ coefficients / (interpolation\ factor)$$

if both the coefficients and inputs are complex

> **Note** The actual accumulator word length may be equal to or greater than the calculated ideal product output word length, depending on the settings on the **Hardware Implementation** pane of the Configuration Parameters dialog box.

- When you select `Same as product output`, these characteristics will match those of the product output.

- When you select `Same as input`, these characteristics will match those of the input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the accumulator, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the accumulator. This block requires power-of-two slope and a bias of zero.

Output

Choose how you will specify the output word length and fraction length:

- When you select `Same as accumulator`, these characteristics will match those of the accumulator.

  A special case occurs when `Inherit via internal rule` is specified for **Accumulator**, and block inputs and coefficients are complex. In that case, the output word length will be one less than the accumulator word length.

- When you select `Same as product output`, these characteristics will match those of the product output.

- When you select `Same as input`, these characteristics will match those of the input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the output, in bits.

FIR Rate Conversion

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the output. This block requires power-of-two slope and a bias of zero.

Lock scaling against changes by the autoscaling tool
Select this parameter to prevent any fixed-point scaling you specify in this block mask from being overriden by the autoscaling tool in the Fixed-Point Settings interface. For more information about the autoscaling tool, refer to "Fixed-Point Settings Interface" on page 8-28.

References [1] Fliege, N. J. *Multirate Digital Signal Processing: Multirate Systems, Filter Banks, Wavelets*. West Sussex, England: John Wiley & Sons, 1994.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point (signed only)
- 8-, 16-, and 32-bit signed integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Downsample	Signal Processing Blockset
FIR Decimation	Signal Processing Blockset
FIR Interpolation	Signal Processing Blockset
Upsample	Signal Processing Blockset
fir1	Signal Processing Toolbox
fir2	Signal Processing Toolbox

firls   Signal Processing Toolbox

upfirdn   Signal Processing Toolbox

See the following sections for related information:

- "Converting Sample and Frame Rates" on page 2-12

- "Multirate Filters" on page 3-66

Flip

Purpose   Flip the input vertically or horizontally

Library   Signal Management / Indexing

Description

The Flip block vertically or horizontally reverses the *M*-by-*N* input matrix, *u*. The output always has the same dimension and frame status as the input.

When you select Columns from the **Flip along** menu, the block *vertically* flips the input so that the first row of the input is the last row of the output.

```
y = flipud(u)      % Equivalent MATLAB code
```

For convenience, length-*M* 1-D vector inputs are treated as *M*-by-1 column vectors for vertical flipping.

When you select Rows from the **Flip along** menu, the block *horizontally* flips the input so that the first column of the input is the last column of the output.

```
y = fliplr(u)      % Equivalent MATLAB code
```

For convenience, length-*N* 1-D vector inputs are treated as 1-by-*N* row vectors for horizontal flipping. The output always has the same dimension and frame status as the input.

This block supports Simulink virtual buses.

Dialog Box
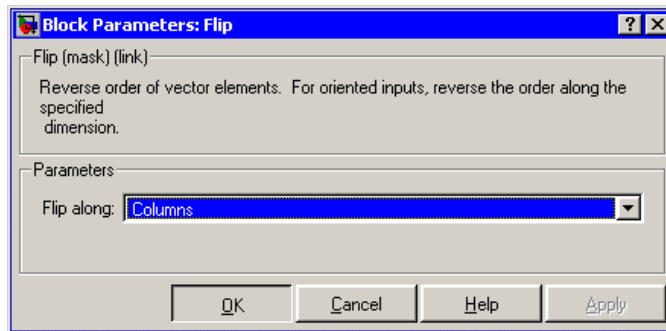
Flip along
> The dimension along which to flip the input. Columns specifies vertical flipping, while Rows specifies horizontal flipping.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Custom data types
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Custom data types
	• Boolean
	• 8-, 16-, and 32-bit signed integers

Flip

Port	Supported Data Types
	• 8-, 16-, and 32-bit unsigned integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Selector	Simulink
Transpose	Signal Processing Blockset
Variable Selector	Signal Processing Blockset
flipud	MATLAB
fliplr	MATLAB

Purpose   Solve *L*X=*B* for X when *L* is lower triangular matrix

Library   Math Functions / Matrices and Linear Algebra / Linear System Solvers

Description   The Forward Substitution block solves the linear system *L*X=*B* by simple forward substitution of variables, where *L* is the lower triangular *M*-by-*M* matrix input to the L port, and *B* is the *M*-by-*N* matrix input to the B port. The output is the solution of the equations, the *M*-by-*N* matrix X, and is always sample based. The block does not check the rank of the inputs.

The block only uses the elements in the *lower triangle* of input *L*; the upper elements are ignored. When you select **Force input to be unit-lower triangular**, the block replaces the elements on the diagonal of *L* with 1's. This is useful when matrix *L* is the result of another operation, such as an LDL decomposition, that uses the diagonal elements to represent the D matrix.

A length-*M* vector input at port B is treated as an *M*-by-1 matrix.

Dialog Box
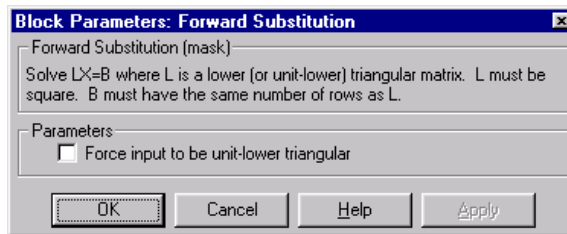
Force input to be unit-lower triangular
Replaces the elements on the diagonal of *L* with 1's when selected. Tunable.

Forward Substitution

Supported Data Types

- Double-precision floating point
- Single-precision floating point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Autocorrelation LPC	Signal Processing Blockset
Cholesky Solver	Signal Processing Blockset
LDL Solver	Signal Processing Blockset
Levinson-Durbin	Signal Processing Blockset
LU Solver	Signal Processing Blockset
QR Solver	Signal Processing Blockset

See "Solving Linear Systems" on page 6-7 for related information.

Purpose   Specify frame status of output signal

Library   Signal Management / Signal Attributes
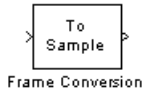
Description   The Frame Conversion block specifies the frame status of the output
signal. Use the **Output signal** parameter to specify the frame status
of the output signal. Your choices are Frame based or Sample based.
The block does not rebuffer or resize two-dimensional inputs. When the
input is a length-M 1-D vector and the **Output signal** parameter is set
to Frame based, the output is a frame-based M-by-1 matrix.

**Dialog
Box**
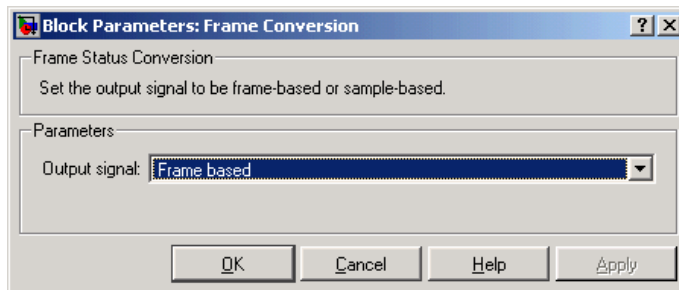
Output signal
Specify the frame status of the output signal.

Frame Conversion

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • Custom data types • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • Custom data types • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Buffer	Signal Processing Blockset
Check Signal Attributes	Signal Processing Blockset
Convert 1-D to 2-D	Signal Processing Blockset
Convert 2-D to 1-D	Signal Processing Blockset
Inherit Complexity	Signal Processing Blockset

Unbuffer	Signal Processing Blockset
Probe	Simulink
Reshape	Simulink
Signal Specification	Simulink

Frame Status Conversion

Purpose   Specify the frame status of the output as sample based or frame based

Library   dspobslib

Description
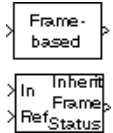
Note The Frame Status Conversion block is still supported but is likely to be obsoleted in a future release. We strongly recommend replacing this block with the Frame Conversion block.

The Frame Status Conversion block passes the input through to the output, and sets the output frame status to the **Output signal** parameter, which can be either Frame-based or Sample-based. The output frame status can also be inherited from the signal at the Ref (reference) input port, which is made visible by selecting the **Inherit output frame status from Ref input port** check box.

When the **Output signal** parameter setting or the inherited signal's frame status differs from the input frame status, the block changes the input frame status accordingly, but does not otherwise alter the signal. In particular, the block does not rebuffer or resize 2-D inputs. Because 1-D vectors cannot be frame based, when the input is a length-M 1-D vector, and the **Output signal** parameter is set to Frame-based, the output is a frame-based M-by-1 matrix (that is, a single channel).

When the **Output signal** parameter or the inherited signal's frame status matches the input frame status, the block passes the input through to the output unaltered.

Dialog Box
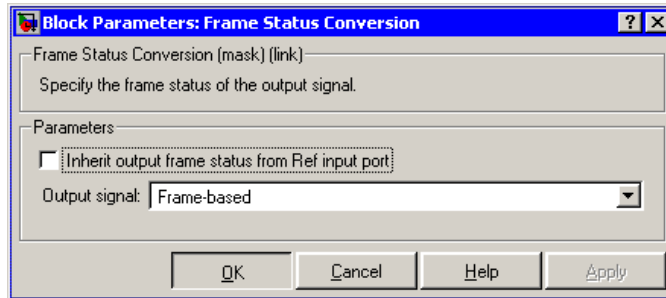
Inherit output frame status from Ref input port
> When selected, enables the Ref input port from which the block inherits the output frame status.

Output signal
> The output frame status, Frame-based or Sample-based.

Supported Data Types

Port	Supported Data Types
In	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed only)
	• Custom data types
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

Frame Status Conversion

Port	Supported Data Types
Ref	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • Custom data types • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • Custom data types • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Check Signal Attributes	Signal Processing Blockset
Convert 1-D to 2-D	Signal Processing Blockset
Convert 2-D to 1-D	Signal Processing Blockset
Inherit Complexity	Signal Processing Blockset

Purpose   Read video frames and/or audio samples from compressed multimedia file

Library   Platform-specific I/O / Windows (WIN32)

Description   The From Multimedia File block reads video frames and/or audio samples from a multimedia file and imports them into a Simulink model. Video processing requires the Video and Image Processing Blockset.

```
speech_dft.mp3
            Audio ▷
A: 22050 Hz, 16 bit, mono
```
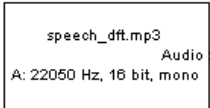
You can view the video frames using a To Video Display block and listen to the audio using a To Wave Device block.

Note This block supports code generation and is only supported on 32-bit Windows platforms. This block performs best on platforms with DirectX Version 9.0 or later and Windows Media Version 9.0 or later.

The output ports of the From Multimedia File block change according the content of the multimedia file. If the file contains video frames, the R, G, and B ports appear on the block. If the file contains audio samples, the Audio port appears on the block.

Port	Output	Supported Data Types	Supports Complex Values?
R, G, B	Matrix that represents one plane of the RGB video stream. Outputs from the R, G, or B port must have same dimensions.	• Double-precision floating point • Single-precision floating point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers	No

From Multimedia File

Port	Output	Supported Data Types	Supports Complex Values?
I	Matrix that represents the intensity video stream	• Double-precision floating point • Single-precision floating point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers	No
Audio	Vector of audio data	• Double-precision floating point • Single-precision floating point • 16-bit signed integers • 8-bit unsigned integers	No

For sink blocks to display video data properly, double- and single-precision floating-point pixel values must be between 0 and 1. For other data types, the pixel values must be between the minimum and maximum values supported by their data type.

Use the **Input file name** parameter to specify the name of the multimedia file from which to read. If the location of this file is on your MATLAB path, enter the filename. If the location of this file is not on your MATLAB path, use the **Browse** button to specify the full path to the file as well as the filename. This parameter also supports URLs.

Use the **Output** parameter to specify whether you want the block to output video frames and/or audio samples. The parameter choices depend on the multimedia file and can include `Video only`, `Audio only`, or `Video and audio`.

If you want the block to output intensity video, select the **Output intensity video** check box.

Use the **Audio output data type** parameter to set the data type of the audio samples output at the Audio port. You can choose double, single, int16, or uint8.

Use the **Video output data type** parameter to set the data type of the video frames output at the R, G, B or I ports. You can choose double, single, int8, uint8, int16, uint16, int32, uint32, or Inherit from file.

Use the **Number of times to play file** parameter to enter the number of times to play the file. The number you enter must be a positive integer or inf.

Use the **Output end-of-file indicator** parameter to determine when the last video frame or audio sample in the multimedia file is output from the block. When you select this check box, a Boolean output port labeled EOF appears on the block. The output from the EOF port is 1 when the last video frame or audio sample is output from the block. Otherwise, the output from the EOF port is 0.

Select the **Inherit sample time from file** check box if you want the sample time of the block to be the same as the sample time of the multimedia file. If you clear this check box, use the **Desired sample time** parameter to specify the block's sample time.

From Multimedia File

Dialog Box
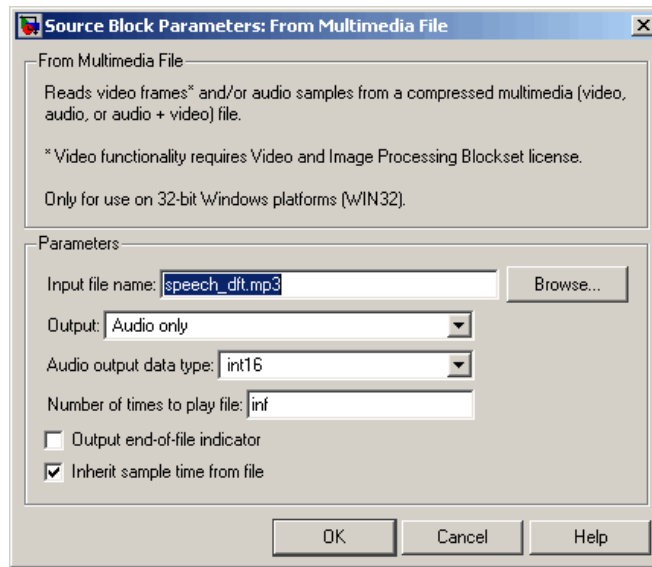
Input file name
> Specify the name of the multimedia file from which to read.

Output
> Specify the block output. The choices depend on the multimedia file and can include Video only, Audio only, or Video and audio.

Output intensity video
> Select this check box if you want the block to output intensity video. This parameter is only available if the multimedia file contains video.

Audio output data type
> Set the data type of the audio samples output at the Audio port. This parameter is only available if the multimedia file contains audio.

Video output data type

Set the data type of the video data output from the block. This parameter is only available if the multimedia file contains video.

Number of times to play file

Enter a positive integer or `inf` to represent the number of times to play the file.

Output end-of-file indicator

Use this check box to determine whether the output is the last video frame or audio sample in the multimedia file.

Inherit sample time from file

Select this check box if you want the sample time of the block to be the same as the sample time of the multimedia file.

Desired sample time

Specify the block's sample time. This parameter is available if you clear the **Inherit sample time from file** check box.

Supported File Formats

Format	Filename Extensions
Apple QuickTime, Macintosh AIFF Resource	`.qt, .aif, .aifc, .aiff, .mov`
Microsoft Windows Media formats	`.avi, .asf, .asx, .rmi, .wav, .wma, .wax, .wmv`
Moving Picture Experts Group (MPEG)	`.mpg, .mpeg, .mlv, .mp2, .mp3, .mpa, .mpe`
UNIX formats	`.au, .snd`

This block supports any multimedia file format supported by Microsoft Windows Media Player. Additionally, this block supports specialized file formats that are associated with any codec supported by Microsoft Windows Media Player.

From Multimedia File

See Also

To Multimedia File	Signal Processing Blockset
From Wave File	Signal Processing Blockset
Image From Workspace	Video and Image Processing Blockset
To Video Display	Video and Image Processing Blockset
Video From Workspace	Video and Image Processing Blockset
Video Viewer	Video and Image Processing Blockset

Purpose   Read audio data from standard audio device in real-time (32-bit
                 Windows operating systems only)

Library   Platform-specific I/O / Windows (WIN32)

Description The From Wave Device block reads audio data from a standard Windows
                 audio device in real-time. It is compatible with most popular Windows
                 hardware, including Sound Blaster cards. (Models that contain both
                 this block and the To Wave Device block require a *duplex-capable* sound
                 card.)

The **Use default audio device** parameter allows the block to detect
and use the system's default audio hardware. This option should be
selected on systems that have a single sound device installed, or when
the default sound device on a multiple-device system is the desired
source. In cases when the default sound device is not the desired input
source, clear **Use default audio device**, and select the desired device
in the **Audio device menu** parameter.

When the audio source contains two channels (stereo), the **Stereo**
check box should be selected. When the audio source contains a single
channel (mono), the **Stereo** check box should be cleared. For stereo
input, the block's output is an M-by-2 matrix containing one frame (M
consecutive samples) of audio data from each of the two channels. For
mono input, the block's output is an M-by-1 matrix containing one
frame (M consecutive samples) of audio data from the mono input. The
frame size, M, is specified by the **Samples per frame** parameter. For
M=1, the output is sample based; otherwise, the output is frame based.

The audio data is processed in uncompressed pulse code modulation
(PCM) format, and should typically be sampled at one of the standard
Windows audio device rates: 8000, 11025, 22050, or 44100 Hz. You can
select one of these rates from the **Sample rate** parameter. To specify a
different rate, select the **User-defined** option and enter a value in the
User-defined sample rate parameter.

The **Sample Width (bits)** parameter specifies the number of bits used
to represent the signal samples read by the audio device. The following
settings are available:

From Wave Device

- 8 — allocates 8 bits to each sample, allowing a resolution of 256 levels

- 16 — allocates 16 bits to each sample, allowing a resolution of 65536 levels

- 24 — allocates 24 bits to each sample, allowing a resolution of 16777216 levels (only for use with 24-bit audio devices)

Higher sample width settings require more memory but yield better fidelity. The output from the block is independent of the **Sample width (bits)** setting. The output data type is determined by the **Data type** parameter setting.

Buffering

Since the audio device accepts real-time audio input, Simulink must read a continuous stream of data from the device throughout the simulation. Delays in reading data from the audio hardware can result in hardware errors or distortion of the signal. This means that the From Wave Device block must read data from the audio hardware as quickly as the hardware itself acquires the signal. However, the block often *cannot* match the throughput rate of the audio hardware, especially when the simulation is running from within Simulink rather than as generated code. (Simulink operations are generally slower than comparable hardware operations, and execution speed routinely varies during the simulation as the host operating system services other processes.) The block must therefore rely on a buffering strategy to ensure that signal data can be read on schedule without losing samples.

At the start of the simulation, the audio device begins writing the input data to a (hardware) buffer with a capacity of T_b seconds. The From Wave Device block immediately begins pulling the earliest samples off the buffer (first in, first out) and collecting them in length-*M* frames for output. As the audio device continues to append inputs to the bottom of the buffer, the From Wave Device block continues to pull inputs off the top of the buffer at the best possible rate.

The following figure shows an audio signal being acquired and output with a frame size of 8 samples. The buffer of the sound board is approaching its five-frame capacity at the instant shown, which means

that the hardware is adding samples to the buffer more rapidly than the block is pulling them off. (If the signal sample rate was 8 kHz, this small buffer could hold approximately 0.005 second of data.

Hardware execution rate is constant.   Simulink execution rate varies.
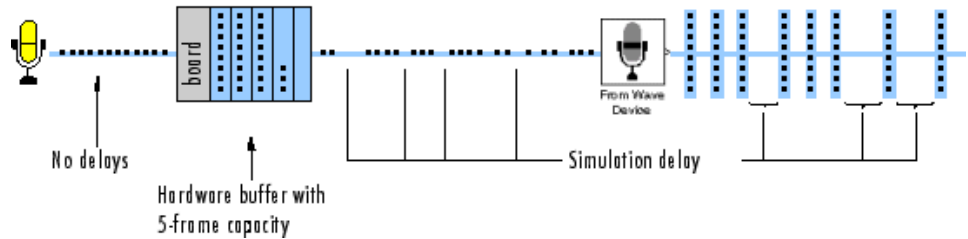
No delays

Hardware buffer with
5-frame capacity

Simulation delay

When the simulation throughput rate is higher than the hardware throughput rate, the buffer remains empty throughout the simulation. If necessary, the From Wave Device block simply waits for new samples to become available on the buffer (the block does not interpolate between samples). More typically, the simulation throughput rate is lower than the hardware throughput rate, and the buffer tends to fill over the duration of the simulation.

Troubleshooting

When the buffer size is too small in relation to the simulation throughput rate, the buffer might fill before the entire length of signal is processed. This usually results in a device error or undesired device output. When this problem occurs, you can choose to either increase the buffer size or the simulation throughput rate:

- *Increase the buffer size*

  The **Queue duration** parameter specifies the duration of signal, T_b (in real-time seconds), that can be buffered in hardware during the simulation. Equivalently, this is the maximum length of time that the block's data acquisition can lag the hardware's data acquisition. The number of frames buffered is approximately

$$\frac{T_b F_s}{M}$$

where F_s is the sample rate of the signal and M is the number of samples per frame. The required buffer size for a given signal depends on the signal length, the frame size, and the speed of the simulation. Note that increasing the buffer size might increase model latency.

- *Increase the simulation throughput rate*

  Two useful methods for improving simulation throughput rates are increasing the signal frame size and compiling the simulation into native code:

  - Increase frame sizes (and convert sample-based signals to frame-based signals) throughout the model to reduce the amount of block-to-block communication overhead. This can drastically increase throughput rates in many cases. However, larger frame sizes generally result in greater model latency due to initial buffering operations.

  - Generate executable code with Real Time Workshop. Native code runs much faster than Simulink, and should provide rates adequate for real-time audio processing.

More general ways to improve throughput rates include simplifying the model, and running the simulation on a faster PC processor. See "Delay and Latency" on page 2-48 and "Improving Simulation Performance and Accuracy" in the Using Simulink documentation for other ideas on improving simulation performance.

Dialog Box
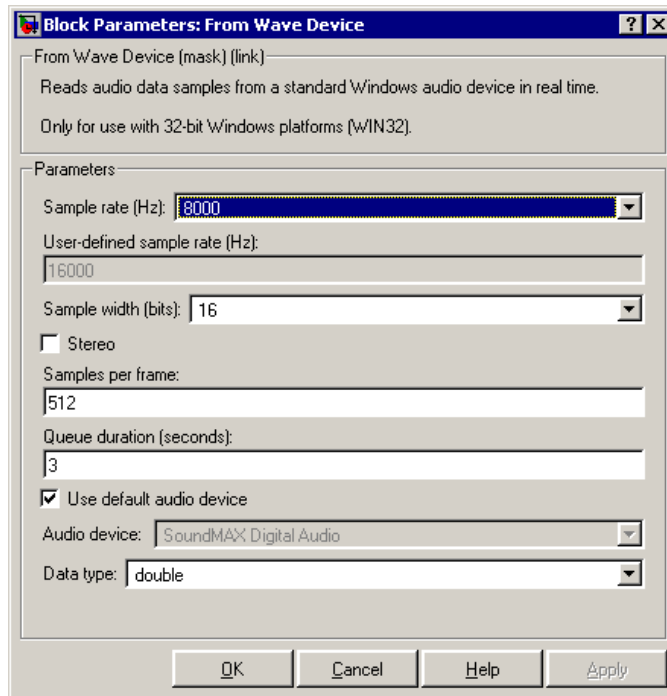
Sample rate (Hz)

The sample rate of the audio data to be acquired. Select one of the standard Windows rates or the User-defined option.

User-defined sample rate (Hz)

The (nonstandard) sample rate of the audio data to be acquired.

Sample width (bits)

The number of bits used to represent each signal sample.

Stereo

Specifies stereo (two-channel) inputs when selected, mono (one-channel) inputs when cleared. Stereo output is M-by-2; mono output is M-by-1.

From Wave Device

Samples per frame
> The number of audio samples in each successive output frame, M.

Queue duration (seconds)
> The length of signal (in seconds) to buffer to the hardware at the start of the simulation.

Use default audio device
> Reads audio input from the system's default audio device when selected. Clear to enable the **Audio device ID** parameter and select a device.

Audio device
> The name of the audio device from which to read the audio output (lists the names of the installed audio device drivers). Select **Use default audio device** when the system has only a single audio card installed.

Data type
> The data type of the output: double-precision, single-precision, signed 16-bit integer, or unsigned 8-bit integer.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- 16-bit signed integer
- 8-bit unsigned integer

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

From Wave File	Signal Processing Blockset
To Wave Device	Signal Processing Blockset
audiorecorder	MATLAB

Purpose   Read audio data from Microsoft Wave (.wav) file

Library   Platform-specific I/O / Windows (WIN32)
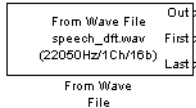
Description   The From Wave File block reads audio data from a Microsoft Wave (.wav) file and generates a signal with one of the data types and amplitude ranges in the following table.

```
From Wave File       Out>
speech_dft.wav      First>
(22050Hz/1Ch/16b)    Last>

   From Wave
     File
```

Output Data Type	Output Amplitude Range
double	±1
single	±1
int16	-32768 to 32767 (-2^{15} to 2^{15} - 1)
uint8	0 to 255

Note This block is supported on 32-bit Windows operating systems only.

The audio data must be in uncompressed pulse code modulation (PCM) format.

```
y = wavread('filename')      % Equivalent MATLAB code
```

The block supports 8-, 16-, 24-, and 32-bit Microsoft Wave (.wav) files.

The **File name** parameter can specify an absolute or relative path to the file. When the file is on the MATLAB path or in the current directory (the directory returned by typing pwd at the MATLAB command line), you need only specify the file's name. You do not need to specify the .wav extension.

When the audio file contains two channels (stereo), the block's output is an *M*-by-2 matrix containing one frame (*M* consecutive samples) of

audio data from each of the two channels. When the audio file contains a single channel (mono), the block's output is an M-by-1 matrix containing one frame (M consecutive samples) of mono audio data. The frame size, M, is specified by the **Samples per output frame** parameter. For M=1, the output is sample based; otherwise, the output is frame based.

The output frame period, T_{fo}, is

$$T_{fo} = \frac{M}{F_s}$$

where F_s is the data sample rate in Hz.

To reduce the required number of file accesses, the block acquires L consecutive samples from the file during each access, where L is specified by the **Minimum number of samples for each read from file** parameter ($L \geq M$). For $L < M$, the block instead acquires M consecutive samples during each access. Larger values of L result in fewer file accesses, which reduces run-time overhead.

Use the **Data type** parameter to specify the data type of the block's output. Your choices are `double`, `single`, `uint8`, or `int16`.

Select the **Loop** check box if you want to play the file more than once. Then, enter the number of times to play the file. The number you enter must be a positive integer or `inf`.

Use the **Number of times to play file** parameter to enter the number of times to play the file. The number you enter must be a positive integer or `inf`, to play the file until you stop the simulation.

The **Samples restart** parameter determines whether the samples from the audio file repeat immediately or repeat at the beginning of the next frame output from the output port. When you select `immediately after last sample`, the samples repeat immediately. When you select `at beginning of next frame`, the frame containing the last sample value from the audio file is zero padded until the frame is filled. The block then places the first sample of the audio file in the first position of the next output frame.

Use the **Output start-of-file indicator** parameter to determine when the first audio sample in the file is output from the block. When you select this check box, a Boolean output port labeled SOF appears on the block. The output from the SOF port is 1 when the first audio sample in the file is output from the block. Otherwise, the output from the SOF port is 0.

Use the **Output end-of-file indicator** parameter to determine when the last audio sample in the file is output from the block. When you select this check box, a Boolean output port labeled EOF appears on the block. The output from the EOF port is 1 when the last audio sample in the file is output from the block. Otherwise, the output from the EOF port is 0.

The block icon shows the name, sample rate (in Hz), number of channels (1 or 2), and sample width (in bits) of the data in the specified audio file. All sample rates are supported; the sample width must be either 8, 16, 24, or 32 bits.

From Wave File

Dialog Box
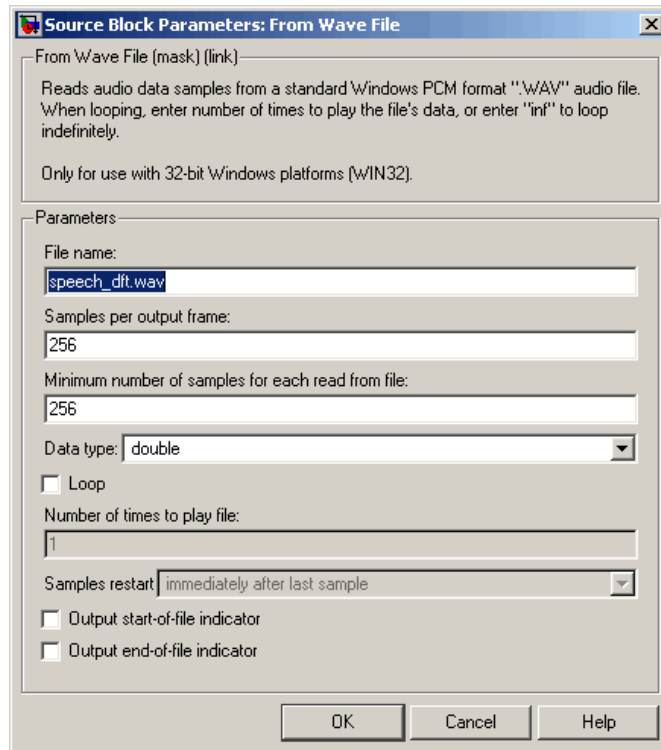
File name
    Enter the path and name of the file to read. Paths can be relative or absolute.

Samples per output frame
    Enter the number of samples in each output frame, M.

Minimum number of samples for each read from file
    Enter the number of consecutive samples to acquire from the file with each file access, L.

Data type

Select the output data type: `double`, `single`, `uint8`, or `int16`. The data type setting determines the output's amplitude range, as shown in the table above.

Loop

Select this check box if you want to play the file more than once.

Number of times to play file

Enter the number of times you want to play the file.

Samples restart

Select `immediately after last sample` to repeat the audio file immediately. Select `at beginning of next frame` to place the first sample of the audio file in the first position of the next output frame.

Output start-of-file indicator

Use this check box to determine whether the output contains the first audio sample in the file.

Output end-of-file indicator

Use this check box to determine whether the output contains the last audio sample in the file.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- 16-bit signed integer
- 8-bit unsigned integer

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

From Wave File

See Also

From Wave Device	Signal Processing Blockset
Signal From Workspace	Signal Processing Blockset
To Wave File	Signal Processing Blockset
wavread	MATLAB

Purpose   Encode linear, pulse code modulation (PCM) narrowband speech signals using A-law or mu-law encoders. Decode index values into quantized output values using A-law or mu-law decoders. Convert between A-law and mu-law index values.

Library   Quantizers
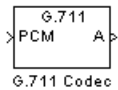
Description   The G711 Codec block is a logarithmic scalar quantizer designed for narrowband speech. Narrowband speech is defined as a voice signal with an analog bandwidth of 4 kHz and a Nyquist sampling frequency of 8 kHz. The block quantizes a narrowband speech input signal so that it can be transmitted using only 8-bits. The G711 Codec block has three modes of operation: encoding, decoding, and conversion. You can choose the block's mode of operation by setting the **Mode** parameter.

```
   G.711
>PCM     A>

G.711 Codec
```

If, for the **Mode** parameter, you choose Encode PCM to A-law, the block assumes that the linear PCM input signal has a dynamic range of 13 bits. Because the block always operates in saturation mode, it assigns any input value above $2^{12} - 1$ to $2^{12} - 1$ and any input value below -2^{12} to -2^{12}. The block implements an A-law quantizer on the input signal and outputs A-law index values. When you choose Encode PCM to mu-law, the block assumes that the linear PCM input signal has a dynamic range of 14 bits. Because the block always operates in saturation mode, it assigns any input value above $2^{13} - 1$ to $2^{13} - 1$ and any input value below -2^{13} to -2^{13}. The block implements a mu-law quantizer on the input signal and outputs mu-law index values.

If, for the **Mode** parameter, you choose Decode A-law to PCM, the block decodes the input A-law index values into quantized output values using an A-law lookup table. When you choose Decode mu-law to PCM, the block decodes the input mu-law index values into quantized output values using a mu-law lookup table.

If, for the **Mode** parameter, you choose Convert A-law to mu-law, the block converts the input A-law index values to mu-law index values. When you choose Convert mu-law to A-law, the block converts the input mu-law index values to A-law index values.

G711 Codec

If, for the **Mode** parameter, you choose `Encode PCM to A-law` or `Encode PCM to mu-law`, the **Overflow diagnostic** parameter appears on the block parameters dialog box. Use this parameter to determine the behavior of the block when overflow occurs. The following options are available:

- `Ignore` — Proceed with the computation and do not issue a warning message.

- `Warning` — Display a warning message in the MATLAB Command Window, and continue the simulation.

- `Error` — Display an error dialog box and terminate the simulation.

Dialog Box

Block Parameters: G.711 Codec

G.711 Codec (mask) (link)

Implements the ITU-T G.711 recommendation for encoding, decoding, or converting speech signals.

The block encodes int16 PCM signals using A-law or mu-law into uint8 codewords. The input is assumed to be a 13-bit (A-law) or 14-bit (mu-law) PCM signal.

The block decodes uint8 codewords into 13-bit (A-law) or 14-bit (mu-law) PCM signals of type int16.

The block converts between A-law and mu-law uint8 codewords.

Parameters

Mode: Encode PCM to A-law

Overflow diagnostic: Warning

| OK | Cancel | Help | Apply |
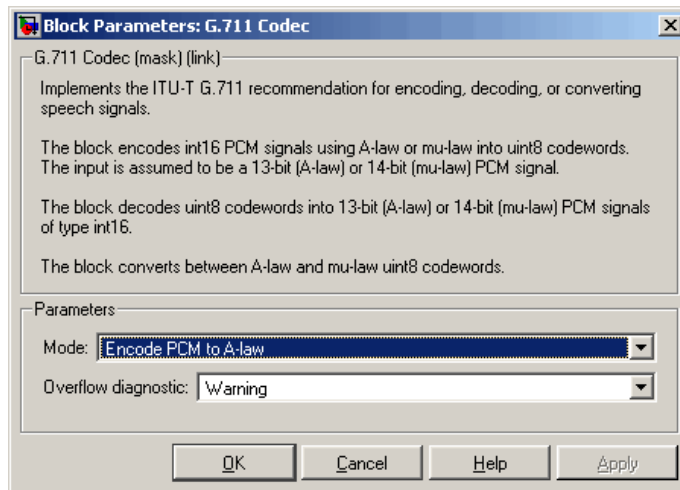
Mode

- When you choose Encode PCM to A-law, the block implements an A-law encoder.

- When you choose Encode PCM to mu-law, the block implements a mu-law encoder.

- When you choose Decode A-law to PCM, the block decodes the input index values into quantized output values using an A-law lookup table.

- When you choose Decode mu-law to PCM, the block decodes the input index values into quantized output values using a mu-law lookup table.

- When you choose Convert A-law to mu-law, the block converts the input A-law index values to mu-law index values.

- When you choose Convert mu-law to A-law, the block converts the input mu-law index values to A-law index values.

G711 Codec

Overflow diagnostic

Use this parameter to determine the behavior of the block when overflow occurs.

- Select `Ignore` to proceed with the computation without a warning message.

- Select `Warning` to display a warning message in the MATLAB Command Window and continue the simulation.

- Select `Error` to display an error dialog box and terminate the simulation.

This parameter is only visible if, for the **Mode** parameter, you select `Encode PCM to A-law` or `Encode PCM to mu-law`.

References ITU-T Recommendation G.711, "Pulse Code Modulation (PCM) of Voice Frequencies," *General Aspects of Digital Transmission Systems*; *Terminal Equipments*, International Telecommunication Union (ITU), 1993.

Supported Data Types

Port	Supported Data Types
PCM	• 16-bit signed integers
A	• 8-bit unsigned integers
mu	• 8-bit unsigned integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Quantizer	Simulink
Scalar Quantizer Decoder	Signal Processing Blockset

Scalar Quantizer Design	Signal Processing Blockset
Uniform Decoder	Signal Processing Blockset
Uniform Encoder	Signal Processing Blockset
Vector Quantizer Decoder	Signal Processing Blockset
Vector Quantizer Design	Signal Processing Blockset
Vector Quantizer Encoder	Signal Processing Blockset

Histogram

Purpose   Generate histogram of input or sequence of inputs

Library   Statistics
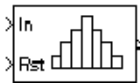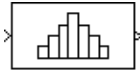
Description   The Histogram block computes the frequency distribution of the elements in each column of the input, or tracks the frequency distribution in a sequence of inputs over a period of time. The **Running histogram** parameter selects between basic operation and running operation, described below. The Histogram block accepts real and complex fixed-point and floating-point inputs.

The block sorts the elements of each column into the number of discrete bins specified by the **Number of bins** parameter, n.

```
y = hist(u,n)      % Equivalent MATLAB code
```

Complex inputs are sorted by magnitude squared.

The histogram value for a given bin represents the *frequency of occurrence* of the input values bracketed by that bin. You specify the upper-boundary of the highest-valued bin in the **Maximum value of input** parameter, B_M, and the lower-boundary of the lowest-valued bin in the **Minimum value of input** parameter, B_m. The bins have equal width of

$$\Delta = \frac{B_M - B_m}{n}$$

and centers located at

$$B_m + \left(k + \frac{1}{2} \right) \Delta \qquad k = 0, 1, 2, \ldots, n - 1$$

Input values that fall on the border between two bins are sorted into the lower-valued bin; that is, each bin includes its upper boundary. For example, a bin of width 4 centered on the value 5 contains the input value 7, but not the input value 3. Input values greater than the **Maximum value of input** parameter or less than **Minimum value of input** parameter are sorted into the highest-valued or lowest-valued

bin, respectively. The values you enter for the **Maximum value of input** and **Minimum value of input** parameters must be real-valued scalars.

Basic Operation

When you do *not* select the **Running histogram** check box, the block computes the frequency distribution of each column in the M-by-N input u independently at each sample time.

For convenience, length-M 1-D vector inputs and *sample-based* length-M row vector inputs are both treated as M-by-1 column vectors.

The output, y, is a sample-based n-by-N matrix whose jth column is the histogram for the data in the jth column of u. When you select the **Normalized** check box, the block scales each column of the output so that sum(y(:,j)) is 1.

Running Operation

When you select the **Running histogram** check box, the block computes the frequency distributions in a *time-sequence* of M-by-N inputs by creating N persistent histograms to which successive inputs are continuously added. For frame-based inputs, this is equivalent to a persistent histogram for each independent channel.

As in basic operation, length-M 1-D vector inputs and *sample-based* length-M row vector inputs are both treated as M-by-1 column vectors.

The output is a sample-based n-by-N matrix whose jth column reflects the current state of the jth histogram. The block resets the running histogram (by emptying all bins of all histograms) when it detects a reset event at the optional Rst port, as described next.

Resetting the Running Histogram

The block resets the running histogram whenever a reset event is detected at the optional Rst port. The reset signal and the input data signal must be the same rate.

To enable the Rst port, select the **Reset port** parameter. You specify the reset event in the **Trigger type** parameter, and can be one of the following:

- Rising edge — Triggers a reset operation when the Rst input does one of the following:

  - Rises from a negative value to a positive value or zero

  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)

- Falling edge — Triggers a reset operation when the Rst input does one of the following:

  - Falls from a positive value to a negative value or zero

  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero (see the following figure)

- `Either edge` — Triggers a reset operation when the Rst input is a Rising edge or Falling edge (as described above)

- `Non-zero sample` — Triggers a reset operation at each sample time that the Rst input is not zero

> **Note** When running simulations in the Simulink `MultiTasking` mode, sample-based reset signals have a one-sample latency, and frame-based reset signals have one frame of latency. Thus, there is a one-sample or one-frame delay between the time the block detects a reset event, and when it applies the reset. For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" on page 2-56 and "Models with Multiple Sample Rates" in the Real-Time Workshop User's Guide documentation.

Examples   This model illustrates the Histogram block's basic operation for a single-channel input, u, where

```
u = [0 -2 6 -12 2 5 4 3 0 4 3 -2 -3 -2 -9]'
```

Histogram
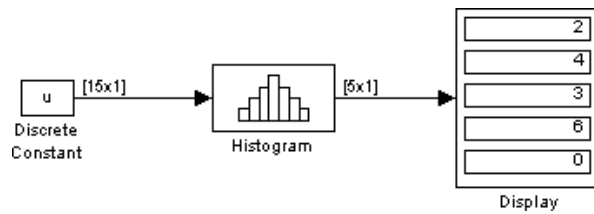
The parameter settings for the Histogram block are

- **Minimum value of input** = -10
- **Maximum value of input** = 10
- **Number of bins** = 5
- **Normalized** = Clear this check box
- **Running histogram** = Clear this check box

The resulting bin width is 4, as shown below.
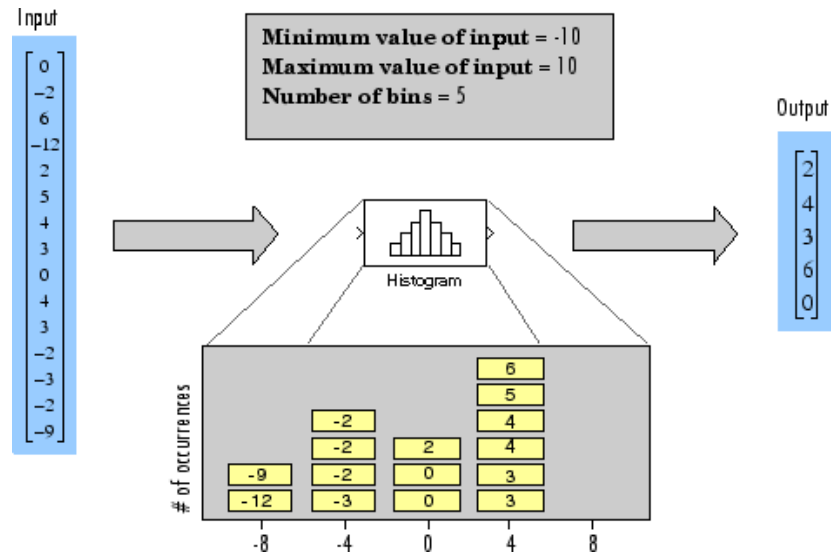
Dialog Box

The **Main** pane of the Histogram block dialog box appears as follows:
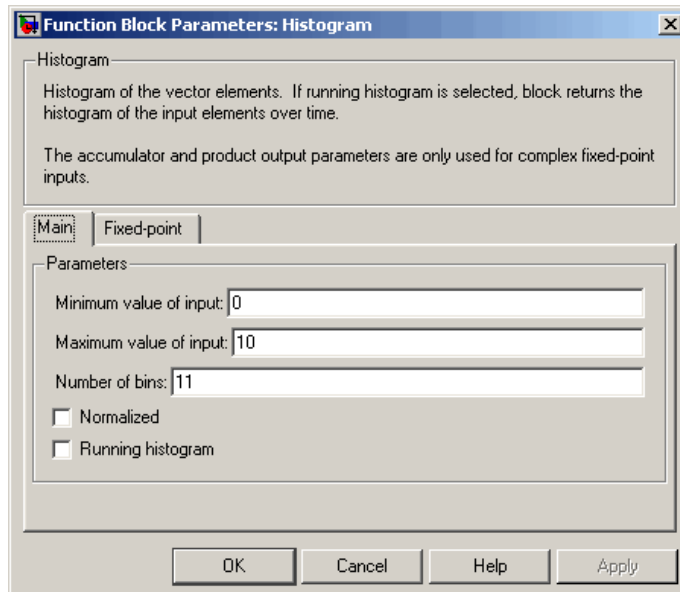
Minimum value of input

Enter a real-valued scalar for the lower boundary, B_m, of the lowest-valued bin. Tunable.

Maximum value of input

Enter a real-valued scalar for the upper boundary, B_M, of the highest-valued bin. Tunable.

Number of bins

The number of bins, n, in the histogram.

Normalized

Normalizes the output vector (1-norm) when selected. Enables running operation when selected. Tunable.

Use of this parameter is not supported for fixed-point signals.

Histogram

Running histogram

Set to enable the running histogram operation, and clear to enable basic histogram operation. For more information, see "Basic Operation" on page 10-505 and "Running Operation" on page 10-505.

Reset port

Enables the `Rst` input port when selected. The reset signal and the input data signal must be the same rate. This parameter is enabled only when you set the **Running histogram** parameter. For more information, see "Running Operation" on page 10-505.

Trigger type

The type of event that resets the running histogram. For more information, see "Resetting the Running Histogram" on page 10-505. This parameter is enabled only when you set the **Reset port** parameter.

The **Fixed-point** pane of the Histogram block dialog box appears as follows:
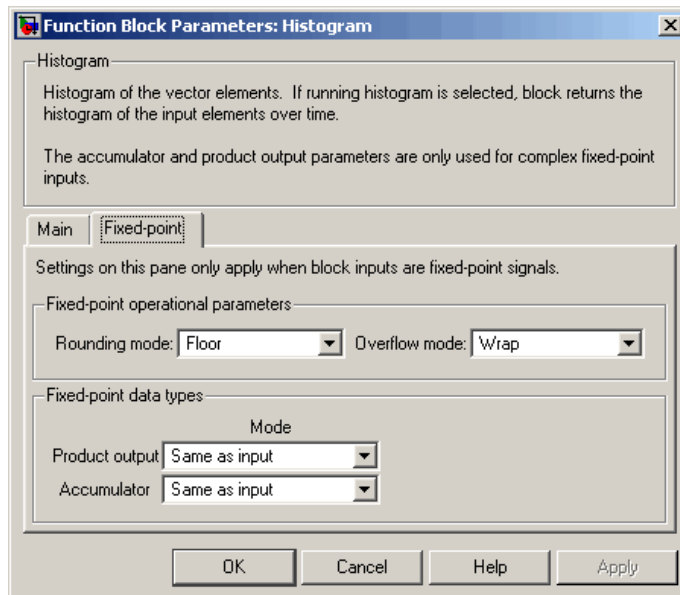
Note The fixed-point parameters listed below are only used for fixed-point complex inputs, which are sorted by squared magnitude.

Rounding mode

Select the rounding mode for fixed-point operations.

Overflow mode

Select the overflow mode for fixed-point operations.

Product output

Use this parameter to specify how you would like to designate the product output word and fraction lengths:

- When you select Same as input, these characteristics will match those of the input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the product output, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the product output. This block requires power-of-two slope and a bias of zero.

Accumulator

Use this parameter to specify the accumulator word and fraction lengths resulting from a complex-complex multiplication in the block:

- When you select `Same as product output`, these characteristics will match those of the product output

- When you select `Same as input`, these characteristics will match those of the input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the accumulator, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the accumulator. This block requires power-of-two slope and a bias of zero.

Supported Data Types

Port	Supported Data Types
In	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

Port	Supported Data Types
Output	• Double-precision floating point
	• Single-precision floating point
	• 32-bit unsigned integers
Rst	• Boolean

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Sort	Signal Processing Blockset
hist	MATLAB

IDCT

Purpose   Compute inverse discrete cosine transform (IDCT) of input

Library   Transforms

Description   The IDCT block computes the inverse discrete cosine transform (IDCT) of each channel in the *M*-by-*N* input matrix, u.

```
y = idct(u)     % Equivalent MATLAB code
```

For both sample-based and frame-based inputs, the block assumes that each input column is a frame containing *M* consecutive samples from an independent channel. The frame size, *M*, must be a power of two. To work with other frame sizes, use the Zero Pad block to pad or truncate the frame size to a power of two length.

The output is an *M*-by-*N* matrix whose *l*th column contains the length-*M* IDCT of the corresponding input column.

$$y(m, l) = \sum_{k=1}^{M} w(k)u(k, l)\cos\frac{\pi(2m-1)(k-1)}{2M}, \qquad m = 1, ..., M$$

where

$$w(k) = \begin{cases} \dfrac{1}{\sqrt{M}}, & k = 1 \\ \sqrt{\dfrac{2}{M}}, & 2 \le k \le M \end{cases}$$

The output is always frame based, and the output sample rate and data type (real/complex) are the same as those of the input.

For convenience, length-*M* 1-D vector inputs and *sample-based* length-*M* row vector inputs are processed as single channels (that is, as *M*-by-1 column vectors), and the output has the same dimension as the input.

The **Sine and cosine computation** parameter determines how the block computes the necessary sine and cosine values. This parameter

has two settings, each with its advantages and disadvantages, as described in the following table.

Sine and Cosine Computation Parameter Setting	Sine and Cosine Computation Method	Effect on Block Performance
Table lookup	The block computes and stores the trigonometric values before the simulation starts, and retrieves them during the simulation. When you generate code from the block, the processor running the generated code stores the trigonometric values computed by the block in a speed-optimized table, and retrieves the values during code execution.	The block usually runs much more quickly, but requires extra memory for storing the precomputed trigonometric values.
Trigonometric fcn	The block computes sine and cosine values during the simulation. When you generate code from the block, the processor running the generated code computes the sine and cosine values while the code runs.	The block usually runs more slowly, but does not need extra data memory. For code generation, the block requires a support library to emulate the trigonometric functions, increasing the size of the generated code.

Fixed-Point Data Types

The diagrams below show the data types used within the IDCT block for fixed-point signals. You can set the sine table, accumulator, product output, and output data types displayed in the diagrams in the IDCT block dialog as discussed in "Dialog Box" on page 10-517.

Inputs to the IDCT block are first cast to the output data type and stored in the output buffer. Each butterfly stage processes signals in

the accumulator data type, with the final output of the butterfly being cast back into the output data type.

Butterfly Stage Data Types

Twiddle Multiplication Data Types

The output of the multiplier is in the product output data type when at least one of the inputs to the multiplier is real. When both of the

inputs to the multiplier are complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed, see "Multiplication Data Types" on page 8-16.

Dialog Box
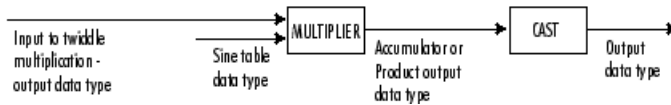
The **Main** pane of the IDCT block dialog appears as follows:

Sine and cosine computation
Sets the block to compute sines and cosines by either looking up sine and cosine values in a speed-optimized table (Table lookup), or by making sine and cosine function calls (Trigonometric fcn). See the table above.

The **Fixed-point** pane of the IDCT block dialog appears as follows:

IDCT

Rounding mode

Select the rounding mode for fixed-point operations. The sine table values do not obey this parameters; they always round to Nearest.

Overflow mode

Select the overflow mode for fixed-point operations. The sine table values do not obey this parameters; they always round to Nearest.

Sine table

Choose how you will specify the word length of the values of the sine table. The fraction length of the sine table values is always equal to the word length minus one:

- When you select Same word length as input, the word length of the sine table values will match that of the input to the block.

- When you select `Specify word length`, you are able to enter the word length of the sine table values, in bits.

  The sine table values do not obey the **Rounding mode** and **Overflow mode** parameters; they are always saturated and rounded to `Nearest`.

Product output

Use this parameter to specify how you would like to designate the product output word and fraction lengths. Refer to "Fixed-Point Data Types" on page 10-515 and "Multiplication Data Types" on page 8-16 for illustrations depicting the use of the product output data type in this block:

- When you select `Inherit via internal rule`, the product output word length and fraction length are automatically set according to the following equations:

  $$product\ output\ word\ length\ =$$
  $$output\ word\ length + sine\ table\ values\ word\ length$$

  $$product\ output\ fraction\ length\ =$$
  $$output\ fraction\ length + sine\ table\ values\ fraction\ length$$

- When you select `Same as input`, these characteristics will match those of the input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the product output, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the product output. This block requires power-of-two slope and a bias of zero.

Accumulator

Use this parameter to specify how you would like to designate the accumulator word and fraction lengths. Refer to "Fixed-Point Data Types" on page 10-515 and "Multiplication Data Types" on

page 8-16 for illustrations depicting the use of the accumulator data type in this block:

- When you select `Inherit via internal rule`, the accumulator word length and fraction length are automatically set according to the following equations:

$$accumulator\ word\ length\ =\ product\ output\ word\ length + 1$$

$$accumulator\ fraction\ length\ =\ product\ output\ fraction\ length$$

- When you select `Same as product output`, these characteristics will match those of the product output.

- When you select `Same as input`, these characteristics will match those of the input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the accumulator, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the accumulator. This block requires power-of-two slope and a bias of zero.

Output

Choose how you will specify the output word length and fraction length:

- When you select `Inherit via internal rule`, the output word length and fraction length are automatically set according to the following equations:

$$output\ word\ length\ =\ input\ word\ length + floor(\log 2(DCT\ length - 1)) + 1$$

$$output\ fraction\ length\ =\ input\ fraction\ length$$

- When you select `Same as input`, these characteristics will match those of the input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the output, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the output. This block requires power-of-two slope and a bias of zero.

Supported Data Types

- Double-precision floating point

- Single-precision floating point

- Fixed point (signed only)

- 8-, 16-, and 32-bit signed integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

DCT	Signal Processing Blockset
IFFT	Signal Processing Blockset
idct	Signal Processing Toolbox

Identity Matrix

Purpose   Generate matrix with ones on the main diagonal and zeros elsewhere

Library   Signal Processing Sources

Math Functions / Matrices and Linear Algebra / Matrix Operations

Description   The Identity Matrix block generates a rectangular matrix with ones on the main diagonal and zeros elsewhere.

When you select the **Inherit output port attributes from input port** check box, the input port is enabled, and an *M*-by-*N* matrix input generates a sample-based *M*-by-*N* matrix output with the same sample period. The *values* in the input matrix are ignored.

```
y = eye([M N])      % Equivalent MATLAB code
```

When you do *not* select the **Inherit output port attributes from input port** check box, the input port is disabled, and the dimensions of the output matrix are determined by the **Matrix size** parameter. A scalar value, M, specifies an *M*-by-*M* identity matrix, while a two-element vector, [*M N*], specifies an *M*-by-*N* unit-diagonal matrix. The output is sample based, and has the sample period specified by the **Sample time** parameter.

Examples   Set **Matrix size** to [3 6] to generate the 3-by-6 unit-diagonal matrix below.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Dialog Box

The **Main** pane of the Identity Matrix block dialog appears as follows:
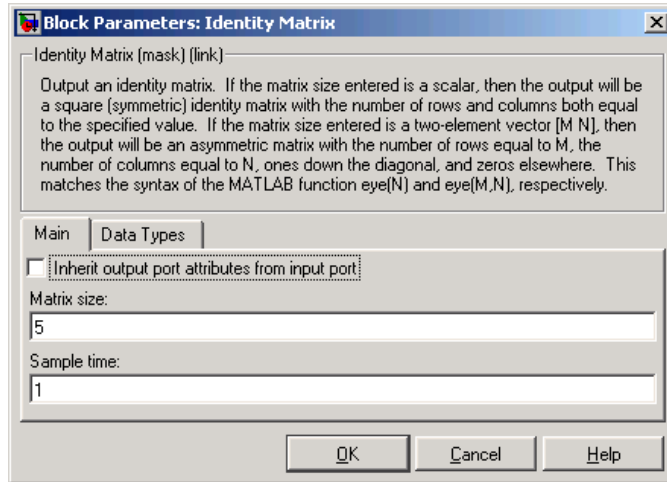
Inherit output port attributes from input port

Enables the input port when selected. The output inherits its dimensions and sample period from the input.

Matrix size

The number of rows and columns in the output matrix: a scalar *M* for a square *M*-by-*M* output, or a vector [*M N*] for an *M*-by-*N* output. This parameter is disabled when you select **Inherit input port attributes from input port**.

Sample time

The discrete sample period of the output. This parameter is disabled when you select **Inherit input port attributes from input port**.

The **Data Types** pane of the Identity Matrix block dialog appears as follows:

Identity Matrix

Output data type

Specify the output data type in one of the following ways:

- Choose one of the built-in data types from the list.

- Choose Fixed-point to specify the output data type and scaling in the **Signed**, **Word length**, **Set fraction length in output to,** and **Fraction length** parameters.

- Choose User-defined to specify the output data type and scaling in the **User-defined data type**, **Set fraction length in output to,** and **Fraction length** parameters.

- Choose Inherit via back propagation to set the output data type and scaling to match the following block

Signed

Select to output a signed fixed-point signal. Otherwise, the signal will be unsigned. This parameter is visible only when you select Fixed-point for the **Output data type** parameter.

Word length

> Specify the word length, in bits, of the fixed-point output data type. This parameter is visible only when you select `Fixed-point` for the **Output data type** parameter.

User-defined data type

> Specify any built-in or fixed-point data type. You can specify fixed-point data types using the `sfix`, `ufix`, `sint`, `uint`, `sfrac`, and `ufrac` functions from Simulink Fixed Point. This parameter is visible only when you select `User-defined` for the **Output data type** parameter.

Set fraction length in output to

> Specify the scaling of the fixed-point output by either of the following two methods:
>
> - Choose `Best precision` to have the output scaling automatically set such that the output signal has the best possible precision.
>
> - Choose `User-defined` to specify the output scaling in the **Fraction length** parameter.
>
> This parameter is visible only when you select `Fixed-point` for the **Output data type** parameter, or when you select `User-defined` and the specified output data type is a fixed-point data type.

Fraction length

> For fixed-point output data types, specify the number of fractional bits, or bits to the right of the binary point. This parameter is visible only when you select `Fixed-point` or `User-defined` for the **Output data type** parameter and `User-defined` for the **Set fraction length in output to** parameter.

Identity Matrix

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point • Custom data types • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point • Custom data types • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Constant Diagonal Matrix	Signal Processing Blockset
DSP Constant	Signal Processing Blockset
eye	MATLAB

Purpose   Compute inverse discrete wavelet transform (IDWT) of input

Library   Transforms

Description

> **Note** The IDWT block is the same as the Dyadic Synthesis Filter Bank
> block in the Multirate Filters library, but with different default settings.
> See the Dyadic Synthesis Filter Bank for more information on how
> to use the block.

The IDWT block computes the inverse discrete wavelet transform
(IDWT) of the input subbands. By default, the block accepts a single
sample-based vector or matrix of concatenated subbands. The output is
frame based, and has the same dimensions as the input. Each column
of the output is the IDWT of the corresponding input column.

You must install the Wavelet Toolbox for the block to automatically
design wavelet-based filters to compute the IDWT. Otherwise, you must
specify your own lowpass and highpass FIR filters by setting the **Filter**
parameter to User defined.

For detailed information about how to use this block, see Dyadic
Synthesis Filter Bank.

Examples   See the examples in the Dyadic Synthesis Filter Bank block reference.

**Supported
Data
Types**
- Double-precision floating point
- Single-precision floating point

To learn how to convert your data types to the above data types in
MATLAB and Simulink, see "Supported Data Types and How to Convert
to Them" on page 7-2.

See Also

Dyadic Synthesis Filter Bank	Signal Processing Blockset
DWT	Signal Processing Blockset

Purpose   Compute inverse fast Fourier transform (IFFT) of input

Library   Transforms

Description   The IFFT block computes the inverse fast Fourier transform (IFFT) of each channel of an *M*-by-*N* or length-*M* input, u, where *M* must be a power of two. To work with other input sizes, use the Zero Pad block to pad or truncate the length-*M* dimension to a power-of-two length.

The output of the IFFT block is equivalent to the MATLAB `ifft` function.

```
y = ifft(u)     % Equivalent MATLAB code
```

The *k*th entry of the *l*th output channel, *y*(*k*, *l*), is equal to the *k*th point of the *M*-point inverse discrete Fourier transform (IDFT) of the *l*th input channel.

$$y(k, l) = \frac{1}{M} \sum_{m=1}^{M} u(m, l) e^{j2\pi(m-1)(k-1)/M} \quad k = 1, \ldots, M$$

This block supports real and complex floating-point and fixed-point inputs.

Input and Output Characteristics

The following table describes valid inputs to the IFFT block, their corresponding outputs, and the dimension along which the block computes the IDFT.

Valid Block Inputs • Real- or complex-valued • M must be a power of two • In linear or bit-reversed order	**Dimension Along Which Block Computes IDFT**	**Corresponding Block Output Characteristics** **Output port rate = input port rate**
Frame-based *M*-by-*N* matrix	Column	The following output characteristics apply to all valid block inputs: • Frame based • Complex valued. If your input is conjugate symmetric and you select the **Input is conjugate symmetric** check box, then the output is real valued. This check box cannot be used for fixed-point signals. • Same dimension as input (for 1-D inputs, output is a length-*M* column) • Each output column (each row for sample-based row inputs) contains the *M*-point IDFT of the corresponding input channel in linear order. If you select the **Skip scaling** check box, the block computes a scaled version of the IDFT that does not include the multiplication factor of 1/*M*.
Sample-based *M*-by-*N* matrix	Column	
Sample-based 1-by-*M* *row vector*	Row	
1-D length-*M* vector	Vector	

Selecting the Twiddle Factor Computation Method

The **Twiddle factor computation** parameter determines how the block computes the necessary sine and cosine terms to calculate the term $e^{j2\pi(m-1)(k-1)/M}$, shown in the first equation of this block reference page. This parameter has two settings, each with its advantages and disadvantages, as described in the following table. Note that only Table lookup mode is supported for fixed-point signals.

Twiddle Factor Computation Parameter Setting	Sine and Cosine Computation Method	Effect on Block Performance
Table lookup	The block computes and stores the trigonometric values before the simulation starts, and retrieves them during the simulation. When you generate code from the block, the processor running the generated code stores the trigonometric values computed by the block, and retrieves the values during code execution.	The block usually runs much more quickly, but requires extra memory for storing the precomputed trigonometric values. You can optimize the table for memory consumption or speed, as described in "Optimizing the Table of Trigonometric Values" on page 10-531.
Trigonometric fcn	The block computes sine and cosine values during the simulation. When you generate code from the block, the processor running the generated code computes the sine and cosine values while the code runs.	The block usually runs more slowly, but does not need extra data memory. For code generation, the block requires a support library to emulate the trigonometric functions, increasing the size of the generated code.

Optimizing the Table of Trigonometric Values

When you set the **Twiddle factor computation** parameter to Table lookup, you need to also set the **Optimize table for** parameter. This parameter optimizes the table of trigonometric values for speed or

memory by varying the number of table entries as summarized in the following table.

Optimize Table for Parameter Setting	Number of Table Entries for N-Point IFFT	Memory Required for Single-Precision 512-Point IFFT
Speed	$3N/4$ — floating point N — fixed point	$\left(\dfrac{3 \times 512}{4}\text{table entries}\right) \times \left(4\dfrac{\text{bytes}}{\text{table entry}}\right)$ $= 1536 \text{ bytes}$
Memory	$N/4$ — floating point Not supported for fixed point	$\left(\dfrac{512}{4}\text{table entries}\right) \times \left(4\dfrac{\text{bytes}}{\text{table entry}}\right)$ $= 512 \text{ bytes}$

Input Order

You must select the **Input is in bit-reversed order** check box to designate whether the ordering of the column elements of the input is linear or bit-reversed order. If you select the **Input is in bit-reversed order** check box, the block assumes the input is in bit-reversed order. If you clear the **Input is in bit-reversed order** check box, block assumes the input is in linear order. For more information ordering of the output, see "Linear and Bit-Reversed Output Order" on page 4-18.

Conjugate Symmetric Input

The FFT block yields conjugate symmetric output when its input is real valued. Taking the IFFT of a conjugate symmetric input matrix produces real-valued output. Therefore, if the input to the block is both floating point and conjugate symmetric and you select the **Input is conjugate symmetric** check box, the block produces real-valued outputs. Selecting this check box optimizes the block's computation method.

If the IFFT block input is conjugate symmetric and you do not select the **Input is conjugate symmetric** check box, the IFFT block outputs a complex-valued signal with small imaginary parts. The block output is invalid if you select this check box and the input is not conjugate symmetric.

Note The **Input is conjugate symmetric** parameter cannot be used for fixed-point signals.

Scaled Output

If you select the **Skip scaling** check box, the block's output is not scaled. If you clear the **Skip scaling** check box and your signal is a floating point signal, the block computes a scaled version of the IDFT, $M \cdot y(k, l)$, which is defined by the following equation.

$$M \cdot y(k, l) = \sum_{m=1}^{M} u(m, l)e^{j2\pi(m-1)(k-1)/M} \quad k = 1, ..., M$$

If you clear the **Skip scaling** check box and your signal is a fixed-point signal, the output of each butterfly of the IFFT is divided by two.

Algorithms Used for IFFT Computation

Depending on whether the block input is floating point or fixed point, real or complex valued, and conjugate symmetric, the block uses one or more of the following algorithms as summarized in the following tables:

- Butterfly operation
- Double-signal algorithm
- Half-length algorithm
- Radix-2 decimation-in-time (DIT) algorithm

For floating-point signals:

Input Complexity	Other Parameter Settings	Algorithms Used for IFFT Computation
Real or complex	☐ Input is in bit-reversed order ☐ Input is conjugate symmetric	Butterfly operation and radix-2 DIT
Real or complex	☑ Input is in bit-reversed order ☐ Input is conjugate symmetric	Radix-2 DIF
Real or complex	☐ Input is in bit-reversed order ☑ Input is conjugate symmetric	Butterfly operation and radix-2 DIT in conjunction with the half-length and double-signal algorithms
Real or complex	☑ Input is in bit-reversed order ☑ Input is conjugate symmetric	Radix-2 DIF in conjunction with the half-length and double-signal algorithms

For fixed-point signals:

Input Complexity	Other Parameter Settings	Algorithms Used for IFFT Computation
Real or complex	☐ Input is in bit-reversed order ☐ Input is conjugate symmetric	Butterfly operation and radix-2 DIT
Real or complex	☑ Input is in bit-reversed order ☐ Input is conjugate symmetric	Radix-2 DIF

Note that the **Input is conjugate symmetric** parameter cannot be used for fixed-point signals.

Fixed-Point Data Types

The diagrams below show the data types used within the IFFT block for fixed-point signals. You can set the sine table, accumulator, product output, and output data types displayed in the diagrams in the IFFT block dialog as discussed in "Dialog Box" on page 10-537.

Inputs to the IFFT block are first cast to the output data type and stored in the output buffer. Each butterfly stage then processes signals in the accumulator data type, with the final output of the butterfly being cast back into the output data type. A twiddle factor is multiplied in before each butterfly stage in a decimation-in-time IFFT, and after each butterfly stage in a decimation-in-frequency IFFT.

IFFT

Decimation-in-time IFFT
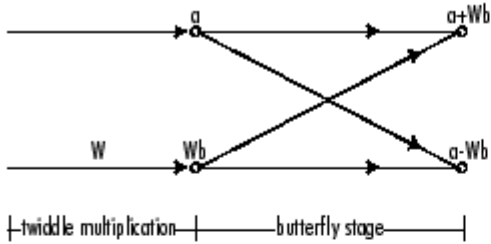
Decimation-in-frequency IFFT

Butterfly stage data types

Twiddle multiplication data types

The output of the multiplier is in the accumulator data type since both of the inputs to the multiplier are complex. For details on the complex multiplication performed, refer to "Multiplication Data Types" on page 8-16.

Dialog Box

The **Main** pane of the IFFT block dialog appears as follows:
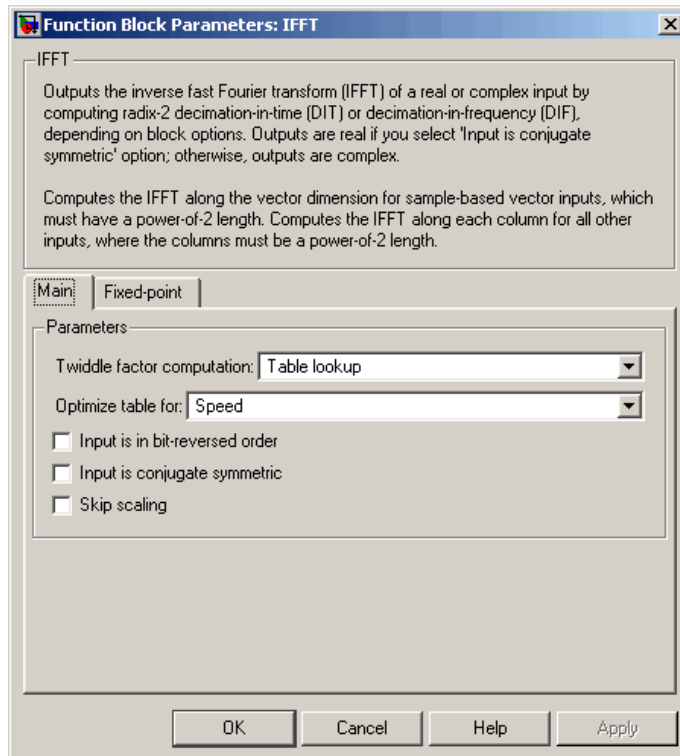
Twiddle factor computation

Specify the computation method of the term $e^{j2\pi(m-1)(k-1)/M}$ shown in the first equation of this block reference page. In Table lookup mode, the block computes and stores the sine and cosine values before the simulation starts. In Trigonometric fcn mode, the block computes the sine and cosine values during the simulation. See "Selecting the Twiddle Factor Computation Method" on page 10-531.

This parameter must be set to `Table lookup` for fixed-point signals.

Optimize table for

Select the optimization of the table of sine and cosine values for `Speed` or `Memory`. This parameter is only available when the **Twiddle factor computation** parameter is set to `Table lookup`. See "Optimizing the Table of Trigonometric Values" on page 10-531.

This parameter must be set to `Speed` for fixed-point signals.

Input is in bit-reversed order

Designate the order of the input channel elements. Select when the input is in bit-reversed order, and clear when the input is in linear order. The block yields invalid outputs when you do not set this parameter correctly. See "Input Order" on page 10-532.

Input is conjugate symmetric

Select when the input to the block is both floating point and conjugate symmetric, and you want real-valued outputs. The block output is invalid when you set this parameter when the input is not conjugate symmetric. This parameter cannot be used for fixed-point signals.

Skip scaling

When you select this check box, no scaling occurs. When this parameter is unselected, scaling does occur:

For floating-point signals, rather than computing the IDFT, the block computes a scaled version of the IDFT. This scaled version of the IDFT does not include the multiplication factor of $1/M$.

For fixed-point signals, the output of each butterfly of the IFFT is divided by two.

The **Fixed-point** pane of the IFFT block dialog appears as follows:
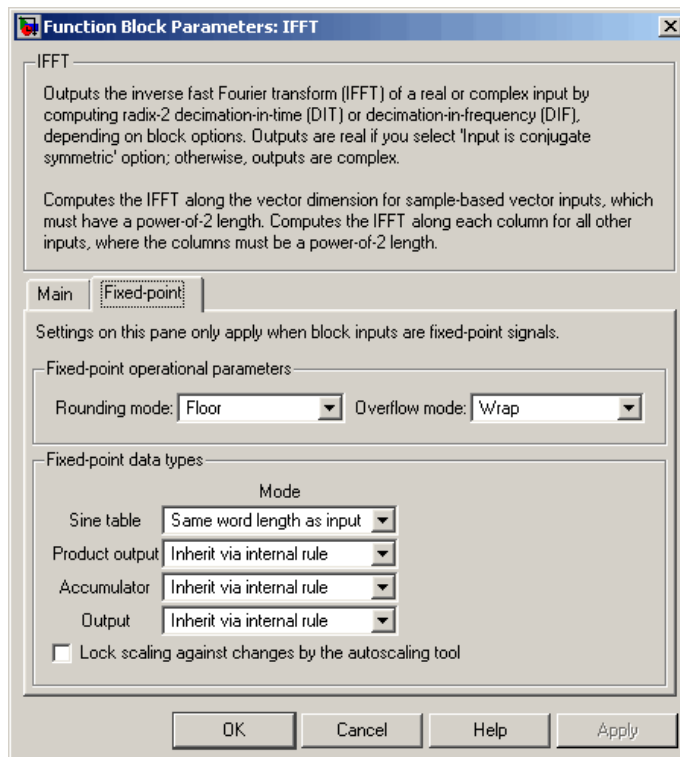
Rounding mode

Select the rounding mode for fixed-point operations. The sine table values do not obey this parameter; they always round to Nearest.

Overflow mode

Select the overflow mode for fixed-point operations. The sine table values do not obey this parameter; they are always saturated.

Sine table

Choose how you will specify the word length of the values of the sine table. The fraction length of the sine table values is always equal to the word length minus one:

- When you select `Same word length as input`, the word length of the sine table values will match that of the input to the block.

- When you select `Specify word length`, you are able to enter the word length of the sine table values, in bits.

The sine table values do not obey the **Rounding mode** and **Overflow mode** parameters; they are always saturated and rounded to `Nearest`.

Product output

Use this parameter to specify how you would like to designate the product output word and fraction lengths. Refer to "Fixed-Point Data Types" on page 10-535 and "Multiplication Data Types" on page 8-16 for illustrations depicting the use of the product output data type in this block:

- When you select `Inherit via internal rule`, the product output word length and fraction length are automatically set according to the following equations:

$$\begin{aligned} &product\ output\ word\ length\ = \\ &output\ word\ length + sine\ table\ values\ word\ length \end{aligned}$$

$$\begin{aligned} &product\ output\ fraction\ length\ = \\ &output\ fraction\ length + sine\ table\ values\ fraction\ length \end{aligned}$$

- When you select `Same as input`, these characteristics will match those of the input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the product output, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the product output. This block requires power-of-two slope and a bias of zero.

Accumulator

Use this parameter to specify how you would like to designate the accumulator word and fraction lengths. Refer to "Fixed-Point Data Types" on page 10-535 and "Multiplication Data Types" on page 8-16 for illustrations depicting the use of the accumulator data type in this block:

- When you select Inherit via internal rule, the accumulator word length and fraction length are automatically set according to the following equations:

$$accumulator\ word\ length\ =\ product\ output\ word\ length + 1$$

$$accumulator\ fraction\ length\ =\ product\ output\ fraction\ length$$

- When you select Same as product output, these characteristics will match those of the product output.

- When you select Same as input, these characteristics will match those of the input to the block.

- When you select Binary point scaling, you are able to enter the word length and the fraction length of the accumulator, in bits.

- When you select Slope and bias scaling, you are able to enter the word length, in bits, and the slope of the accumulator. This block requires power-of-two slope and a bias of zero.

Output

Choose how you will specify the output word length and fraction length:

- When you select Inherit via internal rule, the output word length and fraction length are automatically set according to the following equations:

$$output\ word\ length\ =$$
$$input\ word\ length + \text{floor}(\log2(FFT\ length - 1)) + 1$$

$$output\ fraction\ length\ =\ input\ fraction\ length$$

- When you select `Same as input`, these characteristics will match those of the input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the output, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the output. This block requires power-of-two slope and a bias of zero.

Lock scaling against changes by the autoscaling tool
Select this parameter to prevent any fixed-point scaling you specify in this block mask from being overriden by the autoscaling tool in the Fixed-Point Settings interface. For more information about the autoscaling tool, refer to "Fixed-Point Settings Interface" on page 8-28.

Supported Data Types

- Double-precision floating point

- Single-precision floating point

- Fixed point (signed only)

- 8-, 16-, and 32-bit signed integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

FFT	Signal Processing Blockset
IDCT	Signal Processing Blockset
Pad	Signal Processing Blockset
Zero Pad	Signal Processing Blockset
bitrevorder	Signal Processing Toolbox

fft	Signal Processing Toolbox
ifft	Signal Processing Toolbox

Inherit Complexity

Purpose Change complexity of input to match a reference signal

Library Signal Management / Signal Attributes
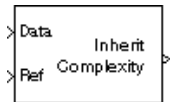
Description The Inherit Complexity block alters the input data at the `Data` port to match the complexity of the reference input at the `Ref` port. When the `Data` input is real, and the `Ref` input is complex, the block appends a zero-valued imaginary component, `0i`, to each element of the `Data` input.
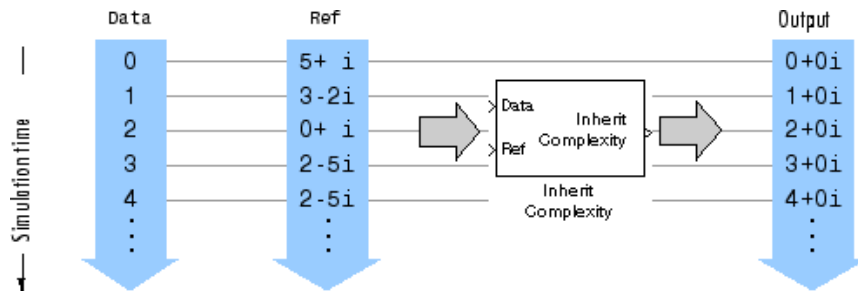
When the `Data` input is complex, and the `Ref` input is real, the block outputs the real component of the `Data` input.
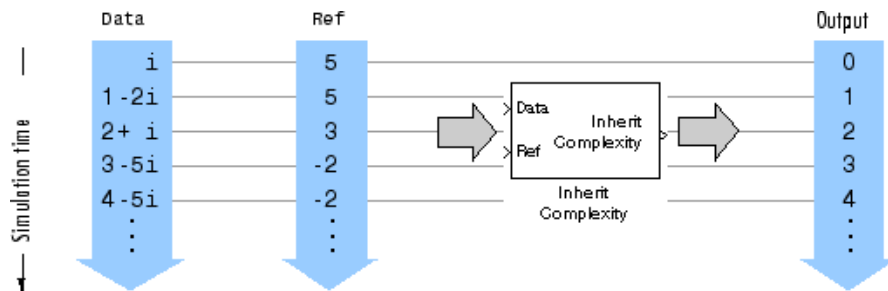
When both the `Data` input and `Ref` input are real, or when both the `Data` input and `Ref` input are complex, the block propagates the `Data` input with no change.

Dialog Box

Block Parameters: Inherit Complexity

Inherit Complexity (mask) (link)

Copy data from the Data input with the complexity of the reference signal. If the data is real and the reference is complex, an all zero imaginary part is created. If the data is complex and the reference is real, the imaginary part of the input is removed.

OK   Cancel   Help   Apply

Supported Data Types

Port	Supported Data Types
Data	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Custom data types
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
Ref	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Custom data types
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point
	• Single-precision floating point

Port	Supported Data Types
	• Fixed point (signed and unsigned)
	• Custom data types
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Check Signal Attributes	Signal Processing Blockset
Complex to Magnitude-Angle	Simulink
Complex to Real-Imag	Simulink
Magnitude-Angle to Complex	Simulink
Real-Imag to Complex	Simulink

Purpose   Delay an input by an integer number of sample periods

Library   dspobslib

Description

z^{-1}
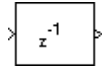
Note The Integer Delay block is still supported but is likely to be obsoleted in a future release. We recommend replacing this block with the Delay block.

The Integer Delay block delays a discrete-time input by the number of sample intervals specified in the **Delay** parameter. Noninteger delay values are rounded to the nearest integer, and negative delays are clipped at 0.

Sample-Based Operation

When the input is a sample-based *M*-by-*N* matrix, the block treats each of the *M*N* matrix elements as an independent channel. The **Delay** parameter, v, can be an *M*-by-*N* matrix of positive integers that specifies the number of sample intervals to delay each channel of the input, or a scalar integer by which to equally delay all channels.

For example, when the input is *M*-by-1 and v is the matrix [v(1) v(2) ... v(M)]', the first channel is delayed by v(1) sample intervals, the second channel is delayed by v(2) sample intervals, and so on. Note that when a channel is delayed for Δ sample-time units, the output sample at time t is the input sample at time $t - \Delta$. When $t - \Delta$ is negative, then the output is the corresponding value specified by the **Initial conditions** parameter.

A 1-D vector of length *M* is treated as an *M*-by-1 matrix, and the output is 1-D.

The **Initial conditions** parameter specifies the output of the block during the initial delay in each channel. The *initial delay* for a particular channel is the time elapsed from the start of the simulation until the first input in that channel is propagated to the output. Both

fixed and time-varying initial conditions can be specified in a variety of ways to suit the dimensions of the input.

Fixed Initial Conditions

A fixed initial condition in sample-based mode can be specified as one of the following:

- *Scalar value* to be repeated at each sample time of the initial delay (for every channel). For a 2-by-2 input with the parameter settings below,

the block generates the following sequence of matrices at the start of the simulation,

$$\begin{bmatrix} -1 & -1 \\ -1 & -1 \end{bmatrix}, \begin{bmatrix} u_{11}^1 & -1 \\ -1 & -1 \end{bmatrix}, \begin{bmatrix} u_{11}^2 & u_{12}^1 \\ -1 & -1 \end{bmatrix}, \begin{bmatrix} u_{11}^3 & u_{12}^2 \\ u_{21}^1 & -1 \end{bmatrix}, \begin{bmatrix} u_{11}^4 & u_{12}^3 \\ u_{21}^2 & u_{22}^1 \end{bmatrix}, \dots$$

where u_{ij}^k is the i,jth element of the kth matrix in the input sequence.

- *Array* of size M-by-N-by-d. In this case, you can set different fixed initial conditions for each element of a sample-based input. This setting is explained further in the *Array* bullet in "Time-Varying Initial Conditions" on page 10-548.

Initial conditions cannot be specified by full matrices.

Time-Varying Initial Conditions

A time-varying initial condition in sample-based mode can be specified in one of the following ways:

- *Vector* of length d, where d is the maximum value specified for any channel in the **Delay** parameter. The vector can be a L-by-d, 1-by-d, or 1-by-1-by-d. The d elements of the vector are output in sequence, one at each sample time of the initial delay.

  For a scalar input and the parameters shown below,

  Delay (samples):
  5
  Initial conditions:
  [-1 -1 -1 0 1]

  the block outputs the sequence `-1,  -1,  -1,  0,  1,...`   at the start of the simulation.

- *Array* of dimension M-by-N-by-d, where d is the value specified for the **Delay** parameter (the *maximum* value when the **Delay** is a vector) and M and N are the number of rows and columns, respectively, in the input matrix. The *d pages* of the array are output in sequence, one at each sample time of the initial delay. For a 2-by-3 input, and the parameters below,

  Delay (samples):
  3
  Initial conditions:
  cat(3, [1 2 3; 4 5 6], [2 4 6; 1 3 5], [3 6 9; 0 4 8])

  the block outputs the matrix sequence

  $$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}, \begin{bmatrix} 2 & 4 & 6 \\ 1 & 3 & 5 \end{bmatrix}, \begin{bmatrix} 3 & 6 & 9 \\ 0 & 4 & 8 \end{bmatrix}$$

  at the start of the simulation. Note that setting **Initial conditions** to an array with the same matrix for each entry implements *constant* initial conditions; a different constant initial condition for each input matrix element (channel).

Initial conditions cannot be specified by full matrices.

Integer Delay

Frame-Based Operation

When the input is a frame-based M-by-N matrix, the block treats each of the N columns as an independent channel, and delays each channel as specified by the **Delay** parameter.

For frame-based inputs, the **Delay** parameter can be a scalar integer by which to equally delay all channels. It can also be a 1-by-N row vector, each element of which serves as the delay for the corresponding channel of the N-channel input. Likewise, it can also be an M-by-1 column vector, each element of which serves as the delay for one of the corresponding M samples for each channel. The **Delay** parameter can be an M-by-N matrix of positive integers as well; in this case, each element of each channel is delayed by the corresponding element in the delay matrix. For instance, if the fifth element of the third column of the delay matrix was 3, then the fifth element of the third channel of the input matrix is always delayed by three sample-time units.

When a channel is delayed for Δ sample-time units, the output sample at time t is the input sample at time $t - \Delta$. When $t - \Delta$ is negative, then the output is the corresponding value specified in the **Initial conditions** parameter.

The **Initial conditions** parameter specifies the output during the initial delay. Both fixed and time-varying initial conditions can be specified. The *initial delay* for a particular channel is the time elapsed from the start of the simulation until the first input in that channel is propagated to the output.

Fixed Initial Conditions

The settings shown below specify *fixed* initial conditions. The value entered in the **Initial conditions** parameter is repeated at the output for each sample time of the initial delay. A fixed initial condition in frame-based mode can be one of the following:

- *Scalar* value to be repeated for all channels of the output at each sample time of the initial delay. For a general M-by-N input with the parameter settings below,

Delay (samples):
5
Initial conditions:
0

the first five samples in each of the *N* channels are zero. Notice that when the frame size is larger than the delay, all of these zeros are all included in the first output from the block.

- *Array* of size 1-by-*N*-by-*D*. In this case, you can also specify different fixed initial conditions for each channel. See the *Array* bullet in "Time-Varying Initial Conditions" on page 10-551 for details.

Initial conditions cannot be specified by full matrices.

Time-Varying Initial Conditions

The following settings specify *time-varying* initial conditions. For time-varying initial conditions, the values specified in the **Initial conditions** parameter are output in sequence during the initial delay. A time-varying initial condition in frame-based mode can be specified in the following ways:

- *Vector* of length *D*, where each of the *N* channels have the same initial conditions sequence specified in the vector. *D* is defined as follows:

  - When an element of the delay entry is less than the frame size,

    $D = d + 1$

    where *d* is the maximum delay.

  - When the all elements of the delay entry are greater than the input frame size,

    $D = d +$ input frame size - 1

  Only the first *d* entries of the initial condition vector will be used; the rest of the values are ignored, but you must include them nonetheless. For a two-channel ramp input `[1:100; 1:100]'` with a frame size of 4 and the parameter settings below,

Delay (samples):
[2 5]
Initial conditions:
[-1 -2 -3 -4 -5 -6]

the block outputs the following sequence of frames at the start of the simulation.

$$\begin{bmatrix} -1 & -1 \\ -5 & -2 \\ 1 & -3 \\ 2 & -4 \end{bmatrix}, \begin{bmatrix} 3 & -5 \\ 4 & 1 \\ 5 & 2 \\ 6 & 3 \end{bmatrix}, \begin{bmatrix} 7 & 4 \\ 8 & 5 \\ 9 & 6 \\ 10 & 7 \end{bmatrix}, \ldots$$

Note that since one of the delays, 2, is less than the frame size of the input, 4, the length of the **Initial conditions** vector is the sum of the maximum delay and 1 (5+1), which is 6. The first five entries of the initial conditions vector are used by the channel with the maximum delay, and the rest of the entries are ignored. Since the first channel is delayed for less than the maximum delay (2 sample time units), it only makes use of two of the initial condition entries.

- *Array* of size 1-by-*N*-by-*D*, where *D* is defined in the *Vector* bullet above in "Time-Varying Initial Conditions" on page 10-551. In this case, the *k*th entry of each 1-by-*N* entry in the array corresponds to an initial condition for the *k*th channel of the input matrix. Thus, a 1-by-*N*-by-*D* initial conditions input allows you to specify different initial conditions for each channel. For instance, for a two-channel ramp input [1:100; 1:100]' with a frame size of 4 and the parameter settings below,

Delay (samples):
5
Initial conditions:
cat(3, [-1 -2], [-3 -4], [-5 -6], [-7 -8], [-9 -10], [0 0], [0 0], [0 0])
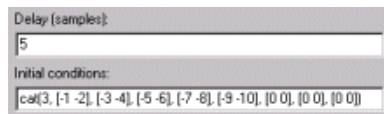
the block outputs the following sequence of frames at the start of the simulation.

$$\begin{bmatrix} -1 & -2 \\ -3 & -4 \\ -5 & -6 \\ -7 & -8 \end{bmatrix}, \begin{bmatrix} -9 & -10 \\ 1 & 1 \\ 2 & 2 \\ 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 \\ 5 & 5 \\ 6 & 6 \\ 7 & 7 \end{bmatrix}, \dots$$

Note that the channels have distinct time varying initial conditions; the initial conditions for channel 1 correspond to the first entry of each length-2 row vector in the initial conditions array, and the initial conditions for channel 2 correspond to the second entry of each row vector in the initial conditions array. Only the first five entries in the initial conditions array are used; the rest are ignored.

The 1-by-N-by-D array entry can also specify different *fixed* initial conditions for every channel; in this case, every 1-by-N entry in the array would be identical, so that the initial conditions for each column are fixed over time.

Initial conditions cannot be specified by full matrices.

Resetting the Delay

The block resets the delay whenever it detects a reset event at the optional Rst port. The reset signal rate must be a positive integer multiple of the rate of the data signal input.

You specify the reset event in the **Reset port** parameter:

- None disables the Rst port.

- Rising edge — Triggers a reset operation when the Rst input does one of the following:

  - Rises from a negative value to a positive value or zero

  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)

- `Falling edge` — Triggers a reset operation when the Rst input does one of the following:

  - Falls from a positive value to a negative value or zero

  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero (see the following figure)

- `Either edge` — Triggers a reset operation when the Rst input is a `Rising edge` or `Falling edge` (as described above).

- `Non-zero sample` — Triggers a reset operation at each sample time that the Rst input is not zero.

Note When running simulations in the Simulink `MultiTasking` mode, sample-based reset signals have a one-sample latency, and frame-based reset signals have one frame of latency. Thus, there is a one-sample or one-frame delay between the time the block detects a reset event, and when it applies the reset. For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" on page 2-56 and "Models with Multiple Sample Rates" in the Real-Time Workshop User's Guide documentation.

Examples   The dspafxr demo illustrates an audio reverberation system built around the Integer Delay block.

Dialog Box

Block Parameters: Integer Delay

Integer Delay (mask) (link)

Delay discrete-time input by a fixed integer number of sample periods.

Parameters

Delay (samples):

1

Initial conditions:

0

Reset port: None

OK   Cancel   Help   Apply
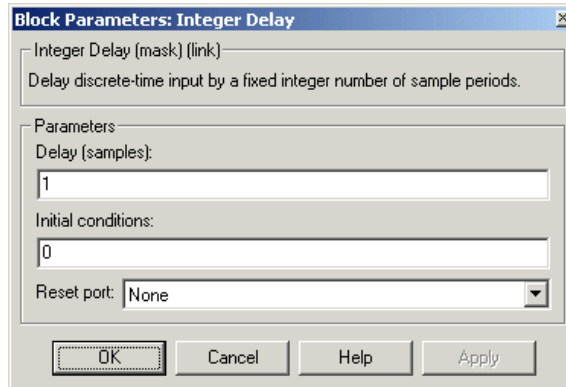
Delay
> The number of sample periods to delay the input signal.

Initial conditions
> The value of the block's output during the initial delay.

Reset port
> Determines the reset event that causes the block to reset the delay. For more information, see "Resetting the Delay" on page 10-553.

Integer Delay

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point (signed only)
- Custom data types
- Boolean — The block accepts Boolean inputs to the Rst port, which is enabled by the **Reset port** parameter.
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Unit Delay	Simulink
Variable Fractional Delay	Signal Processing Blockset
Variable Integer Delay	Signal Processing Blockset

Purpose　　　Interpolate values of real input samples

Library　　　Signal Operations
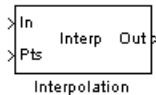
Description　　The Interpolation block interpolates each channel of discrete, real, inputs using linear or FIR interpolation. The input can be a sample or frame based vector or matrix. The output is a vector or matrix of the interpolated values, and has the same frame status and frame rate as the input.

You must specify the *interpolation points* (times at which to interpolate values) in an *interpolation vector*, I_n. An entry of 1 in I_n refers to the first sample of the input, an entry of 2.5 refers to the sample half-way between the second and third input sample, and so on. I_n must have the same frame status and frame rate as the input, and can be a length-*P* row or column vector, where *P* is usually any positive integer.

Usually, the block applies the vector I_n to each column of an input matrix, or to each input vector. You can set the block to either apply the *same* interpolation vector for all input vectors or matrices (*static* interpolation points), or use a *different* interpolation vector for each input vector or matrix (*time-varying* interpolation points).

For more information, see other sections of this reference page.

Sections of This Reference Page

- "Specifying Static Interpolation Points" on page 10-558
- "Specifying Time-Varying Interpolation Points" on page 10-558
- "How the Block Applies Interpolation Vectors to Inputs" on page 10-558
- "Handling Out-of-Range Interpolation Points" on page 10-561
- "Linear Interpolation Mode" on page 10-562
- "FIR Interpolation Mode" on page 10-563
- "Dialog Box" on page 10-564

Interpolation

- "Supported Data Types" on page 10-566

Specifying Static Interpolation Points

To supply the block with a *static* interpolation vector (an interpolation vector applied to every input vector or matrix), do the following:

- Set the **Source of interpolation points** parameter to `Specify via dialog`.

- Enter the interpolation vector in the **Interpolation points** parameter. To learn about interpolation vectors, see "How the Block Applies Interpolation Vectors to Inputs" on page 10-558.

Specifying Time-Varying Interpolation Points

To supply the block with time-varying interpolation vectors (where the block uses a different interpolation vector for each input vector or matrix), do the following:

1 Set the **Source of interpolation points** parameter to `Input port`, the Pts port appears on the block.

2 Generate a signal of interpolation vectors with the *same frame status* and *same frame rate* as the input signal, and supply it to the Pts port. The block uses the input to this port as the interpolation points. To learn about interpolation vectors, see "How the Block Applies Interpolation Vectors to Inputs" on page 10-558.

How the Block Applies Interpolation Vectors to Inputs

The interpolation vector I_n represents the points in time at which to interpolate values of the input signal. An entry of 1 in I_n refers to the first sample of the input, an entry of 2.5 refers to the sample half-way between the second and third input sample, and so on. In most cases, the vector I_n can be of any length.

Depending on the dimension and frame status of the input and the dimension of I_n, the block usually applies I_n to the input in one of the following ways:

- Applies the vector I_n to each channel of a matrix input, resulting in a matrix output.

- Applies the vector I_n to each input vector (as if the input vector were a single channel), resulting in a vector output with the same orientation as the input (row or column).

The following tables summarize how the block applies the vector I_n to all the possible types of sample- and frame-based inputs, and show the resulting output dimensions. (The block applies both static and time-varying interpolation vectors to the input signal in the same way.)

How Block Applies Interpolation Vectors to Frame-Based Inputs

Frame-Based Input Dimensions	Dimensions of Interpolation Vector I_n (P is a positive integer)	How Block Applies I_n to Input	Frame-Based Output Dimensions
M-by-N matrix	P-by-1 column	Applies I_n to each input column	P-by-N matrix
	1-by-N row	Applies each column of I_n (each element of I_n) to the corresponding columns of the input	1-by-N row
M-by-1 column	P-by-1 column	Applies I_n to the input column	P-by-1 column
	1-by-P row (block treats as a column)	Applies I_n to the input column	P-by-1 column

Interpolation

Frame-Based Input Dimensions	Dimensions of Interpolation Vector I_n (P is a positive integer)	How Block Applies I_n to Input	Frame-Based Output Dimensions
1-by-N row (not recommended)	P-by-1 column	not applicable	P-by-N matrix where each row is a copy of the input vector
	1-by-P row	not applicable	1-by-N row, a copy of the input vector

How Block Applies Interpolation Vectors to Sample-Based Inputs

Sample-Based Input Dimensions	Dimensions of Interpolation Vector I_n (P is any positive integer)	How Block Applies I_n to Input	Sample-Based Output Dimensions
M-by-N matrix	P-by-1 column	Applies I_n to each input column	P-by-N matrix
	1-by-P row (block treats as a column)	Applies I_n to each input column	P-by-N matrix
M-by-1 column	P-by-1 column	Applies I_n to the input column	P-by-1 column
	1-by-P row (block treats as a column)	Applies I_n to the input column	P-by-1 column
1-by-N row	P-by-1 column (block treats as a row)	Applies I_n to the input row	1-by-P row
	1-by-P row	Applies I_n to the input row	1-by-P row

Handling Out-of-Range Interpolation Points

The *valid range* of the values in the interpolation vector I_n is from 1 to the number of samples in each channel of the input. For instance, given a length-5 input vector D, all entries of I_n must range from 1 to 5. I_n cannot contain entries such as 7 or -9, since there is no 7th or -9th entry in D.

The **Out of range interpolation points** parameter sets how the block handles interpolation points that are not within the valid range, and has the following settings:

- Clip — The block replaces any out-of-range values in I_n with the closest value in the valid range (from 1 to the number of input samples), and then proceeds with computations using the clipped version of I_n.

- Clip and warn — In addition to Clip, the block issues a warning at the MATLAB command line every time clipping occurs.

- Error — When the block encounters an out-of-range value in I_n, the simulation stops and the block issues an error at the MATLAB command line.

Example of Clipping

Suppose the block is set to clip out-of-range interpolation points, and gets the following input vector and interpolation points:

- D = [11, 22, 33, 44]'

- I_n = [10, 2.6, -3]'

Since *D* has four samples, valid interpolation points range from 1 to 4. The block clips the interpolation point 10 to 4 and the point -3 to 1, resulting in the clipped interpolation vector $I_{nclipped}$ = [4, 2.6, 1]'.

Interpolation

Linear Interpolation Mode

When **Interpolation Mode** is set to `Linear`, the block interpolates data values by assuming that the data varies linearly between samples taken at adjacent sample times.

For instance, if the input signal D = [1, 2, 1.5, 3, 0.25]', the following left-hand plot shows the samples in D, and the right-hand plot shows the linearly interpolated values between the samples in D.
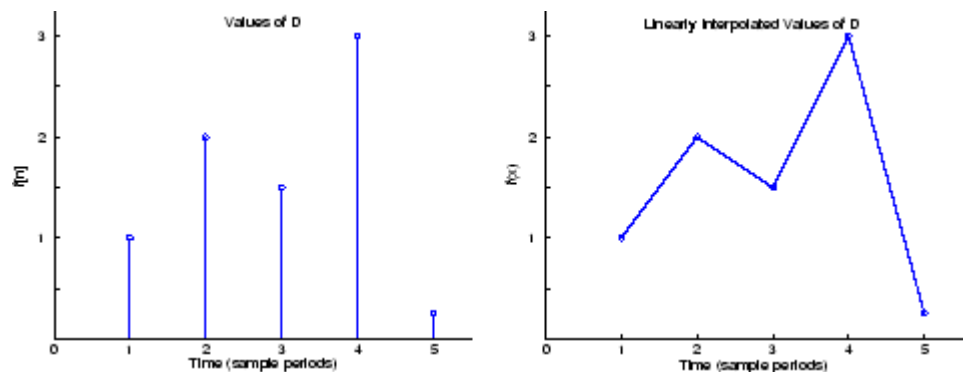
As illustrated below, if the block is in linear interpolation mode and is set to clip out-of-range interpolation points, where

- D = [1, 2, 1.5, 3, 0.25]'

- I_n = [-4, 2.7, 4.3, 10]'

then the block clips the invalid interpolation points, and outputs the linearly interpolated values in a vector, [1, 1.65, 2.175, 0.25]'.

Interpolated Values of D at Clipped Interpolation Times
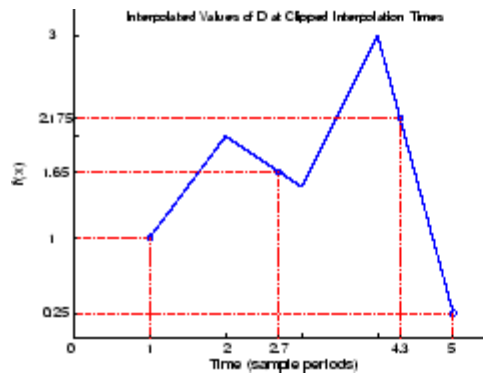
D = [1, 2, 1.5, 3, 0.25]'

I_n = [-4, 2.7, 4.3, 10]'

The valid time range is from 1 to 5 sample periods, so -4 is clipped to 1, and 10 is clipped to 5.

Clipped I_n = [1, 2.7, 4.3, 5]'

FIR Interpolation Mode

When **Interpolation Mode** is set to FIR, the block interpolates data values using an FIR interpolation filter, specified by various block parameters. See "FIR Interpolation Mode" on page 10-1132 in the Variable Fractional Delay block reference for more information.

Interpolation

Dialog Box
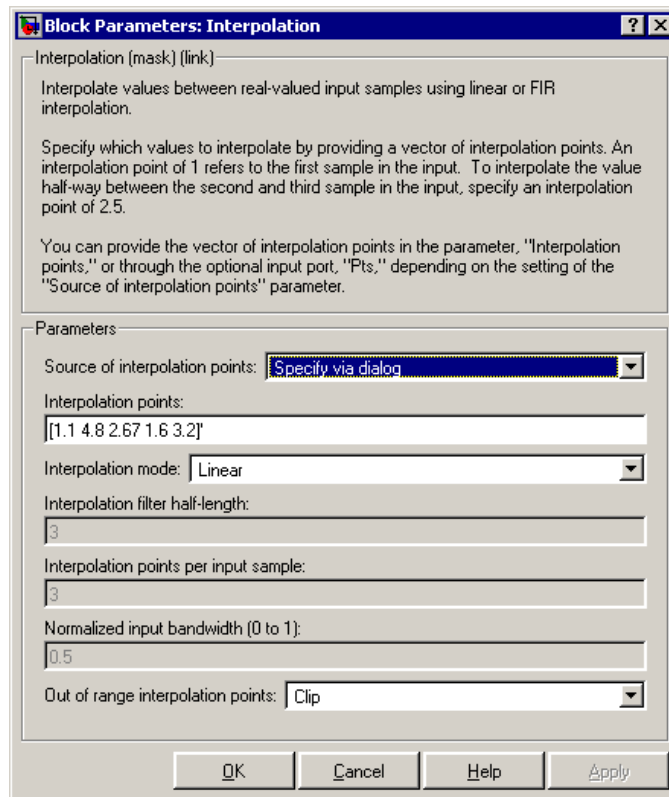
Source of interpolation points

Choose how you want to specify the interpolation points. If you select Specify via dialog, the **Interpolation points** parameter become available. Use this option for static interpolation points. If you select Input port, the Pts port appears on the block. The block uses the input to this port as the interpolation points. Use this option for time-varying interpolation points. For more information, see "Specifying Static Interpolation Points" on page 10-558 and "Specifying Time-Varying Interpolation Points" on page 10-558. Nontunable.

Interpolation points

The vector I_n of points in time at which to interpolate the input signal. An entry of 1 in I_n refers to the first sample of the input, an entry of 2.5 refers to the sample half-way between the second and third input sample, and so on. See "How the Block Applies Interpolation Vectors to Inputs" on page 10-558. Tunable.

Interpolation mode

Sets the block to interpolate by either linear or FIR interpolation. For more information, see "Linear Interpolation Mode" on page 10-562 and "FIR Interpolation Mode" on page 10-563. Nontunable.

Interpolator filter half-length

Half the length of the FIR interpolation filter. For more information, see "FIR Interpolation Mode" on page 10-563. Nontunable.

Interpolation points per input sample

The number Q, where the FIR interpolation filter uses the nearest 2*Q points in the signal to interpolate the value at an interpolation point. When there are less than 2*Q neighboring points, the block uses linear interpolation in place of FIR interpolation. For more information, see "FIR Interpolation Mode" on page 10-563. and "Linear Interpolation Mode" on page 10-562. Nontunable.

Normalized input bandwidth (0 to 1)

The bandwidth of the input divided by Fs/2 (half the input sample frequency). For more information, see "FIR Interpolation Mode" on page 10-563. Nontunable.

Out of range interpolation points

When an interpolation point is out of range, this parameter sets the block to either clip the interpolation point, clip the value and issue a warning at the MATLAB command line, or stop the simulation and issue an error at the MATLAB command line. For more information, see "Handling Out-of-Range Interpolation Points" on page 10-561. Nontunable.

Interpolation

Supported Data Types

Port	Supported Data Types
In	• Double-precision floating point • Single-precision floating point
Pts	• Double-precision floating point • Single-precision floating point
Out	• Double-precision floating point • Single-precision floating point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

Purpose

Recover time-domain signals by performing an inverse short-time, fast Fourier transform (FFT)

Library

Transforms

Description

The Inverse Short-Time FFT block reconstructs the time-domain signal from the frequency-domain output of the Short-Time FFT block using a two-step process. First, the block performs the overlap add algorithm shown below.

$$x[n] = \frac{L}{W(0)} \sum_{p = -\infty}^{\infty} \left[\frac{1}{N} \sum_{k = 0}^{N-1} X[pL, k] e^{j2\pi kn/N} \right]$$

Then, the block rebuffers the signal in order to reconstruct the frame-based time-domain signal. Depending on the analysis window used by the Short-Time FFT block, the Inverse Short-Time FFT block might or might not achieve perfect reconstruction of the time domain signal.

Connect your complex-valued, sample-based, single-channel or multichannel input signal to the X(n,k) port. The block uses the **Overlap between consecutive STFFT frames (in samples)** and **Samples per output frame** parameters as well as the **Input is conjugate symmetric** check box to reconstruct the original time-domain signal. The real or complex-valued, frame-based, single-channel or multichannel inverse short-time FFT is output at port x(n).

Connect your complex-valued, sample-based or frame-based, single-channel analysis window to the w(n) port. When you select the **Assert if analysis window does not support perfect signal reconstruction** check box, the block displays an error when the input signal cannot be perfectly reconstructed. The block uses the values you enter for the **Analysis window length (W)** and **Reconstruction error tolerance**, or maximum amount of allowable error in the reconstruction process, to determine if the signal can be perfectly reconstructed.

Inverse Short-Time FFT

Examples

The dspstsa demo illustrates how to use the Short-Time FFT and Inverse Short-Time FFT blocks to remove the background noise from a speech signal.

Dialog Box
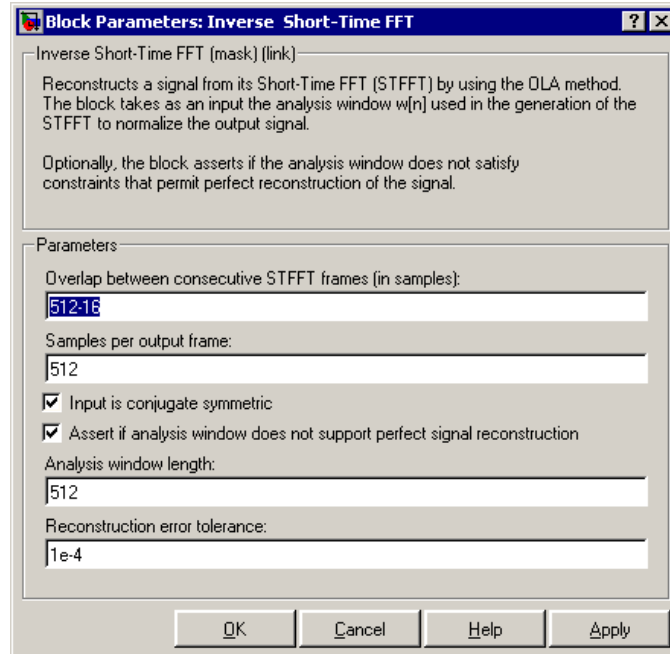
Overlap between consecutive STFFT frames (in samples)

Enter the number of samples of overlap for each frame of the Short-Time FFT block's input signal. This value should be the same as the **Overlap between consecutive windows (in samples)** parameter in the Short-Time FFT block parameters dialog box.

Samples per output frame

Enter the desired frame length of the frame-based output signal.

Input is conjugate symmetric

Select this check box when the input to the block is both floating point and conjugate symmetric, and you want real-valued outputs. When you select this check box when the input is not conjugate symmetric, the output of the block is invalid. This parameter cannot be used for fixed-point signals.

Assert if analysis window does not support perfect signal reconstruction

Select this check box to display an error when the analysis window used by the Short-Time FFT block does not support perfect signal reconstruction.

Analysis window length

Enter the length of the analysis window. This parameter is visible when you select the **Assert if analysis window does not support perfect signal reconstruction** check box.

Reconstruction error tolerance

Enter the amount of acceptable error in the reconstruction of the original signal. This parameter is visible when you select the **Assert if analysis window does not support perfect signal reconstruction** check box.

References   Quatieri, Thomas E. *Discrete-Time Speech Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 2001.

Supported Data Types

Port	Supported Data Types
X(n,k)	• Double-precision floating point
	• Single-precision floating point
w(n)	• Double-precision floating point
	• Single-precision floating point
x(n)	• Double-precision floating point
	• Single-precision floating point

Inverse Short-Time FFT

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Burg Method	Signal Processing Blockset
Magnitude FFT	Signal Processing Blockset
Periodogram	Signal Processing Blockset
Short-Time FFT	Signal Processing Blockset
Spectrum Scope	Signal Processing Blockset
Window Function	Signal Processing Blockset
Yule-Walker Method	Signal Processing Blockset
pwelch	Signal Processing Toolbox

Purpose   Compute filter estimates for inputs using Kalman adaptive filter
algorithm

Library   Filtering / Adaptive Filters

Description   The Kalman Adaptive Filter block computes the optimal linear
minimum mean-square estimate (MMSE) of the FIR filter coefficients
using a one-step predictor algorithm. This Kalman filter algorithm is
based on the following physical realization of a dynamic system.
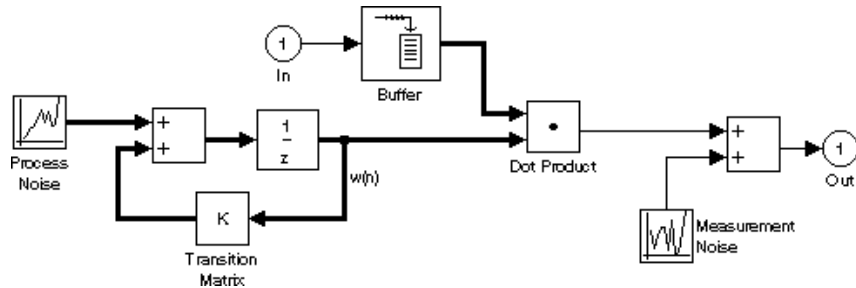
The Kalman filter assumes that there are no deterministic changes to
the filter taps over time (that is, the transition matrix is identity), and
that the only observable output from the system is the filter output
with additive noise. The corresponding Kalman filter is expressed in
matrix form as

$$g(n) = \frac{K(n-1)u(n)}{u^H(n)K(n-1)u(n) + Q_M}$$

$$y(n) = u^H(n)\hat{w}(n)$$

$$e(n) = d(n) - y(n)$$

$$\hat{w}(n+1) = \hat{w}(n) + e(n)g(n)$$

$$K(n) = K(n-1) - g(n)u^H(n)K(n-1) + Q_P$$

The variables are as follows

Kalman Adaptive Filter

Variable	Description
n	The current algorithm iteration
$u(n)$	The buffered input samples at step n
$K(n)$	The correlation matrix of the state estimation error
$g(n)$	The vector of Kalman gains at step n
$\hat{w}(n)$	The vector of filter-tap estimates at step n
$y(n)$	The filtered output at step n
$e(n)$	The estimation error at step n
$d(n)$	The desired response at step n
Q_{M}	The correlation matrix of the measurement noise
Q_{P}	The correlation matrix of the process noise

The correlation matrices, Q_{M} and Q_{P}, are specified in the parameter dialog box by scalar variance terms to be placed along the matrix diagonals, thus ensuring that these matrices are symmetric. The filter algorithm based on this constraint is also known as the *random-walk Kalman filter*.

The implementation of the algorithm in the block is optimized by exploiting the symmetry of the input covariance matrix $K(n)$. This decreases the total number of computations by a factor of two.

The block icon has port labels corresponding to the inputs and outputs of the Kalman algorithm. Note that inputs to the In and Err ports must be sample-based scalars with the same complexity. The signal at the Out port is a scalar, while the signal at the Taps port is a sample-based vector.

Block Ports	Corresponding Variables
In	u, the scalar input, which is internally buffered into the vector $u(n)$
Out	$y(n)$, the filtered scalar output
Err	$e(n)$, the scalar estimation error
Taps	$\hat{w}(n)$, the vector of filter-tap estimates

An optional Adapt input port is added when you select the **Adapt port** check box in the dialog box. When this port is enabled, the block continuously adapts the filter coefficients while the Adapt input is nonzero. A zero-valued input to the Adapt port causes the block to stop adapting, and to hold the filter coefficients at their current values until the next nonzero Adapt input.

The **FIR filter length** parameter specifies the length of the filter that the Kalman algorithm estimates. The **Measurement noise variance** and the **Process noise variance** parameters specify the correlation matrices of the measurement and process noise, respectively. The **Measurement noise variance** must be a scalar, while the **Process noise variance** can be a vector of values to be placed along the diagonal, or a scalar to be repeated for the diagonal elements.

The **Initial value of filter taps** specifies the initial value $\hat{w}(0)$ as a vector, or as a scalar to be repeated for all vector elements. The **Initial error correlation matrix** specifies the initial value $K(0)$, and can be a diagonal matrix, a vector of values to be placed along the diagonal, or a scalar to be repeated for the diagonal elements.

Kalman Adaptive Filter

**Dialog
Box**

FIR filter length
> The length of the FIR filter.

Measurement noise variance
> The value to appear along the diagonal of the measurement noise
> correlation matrix. Tunable.

Process noise variance
> The value to appear along the diagonal of the process noise
> correlation matrix. Tunable.

Initial value of filter taps
> The initial FIR filter coefficients.

Initial error correlation matrix
    The initial value of the error correlation matrix.

Adapt port
    Enables the Adapt port.

References Haykin, S. *Adaptive Filter Theory*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1996.

Supported Data Types
- Double-precision floating point
- Single-precision floating point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

LMS Adaptive Filter	Signal Processing Blockset
RLS Adaptive Filter	Signal Processing Blockset

See "Adaptive Filters" on page 3-53 for related information.

LDL Factorization

Purpose   Factor square Hermitian positive definite matrices into lower, upper, and diagonal components

Library   Math Functions / Matrices and Linear Algebra / Matrix Factorizations
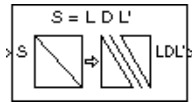
Description   The LDL Factorization block uniquely factors the square Hermitian positive definite input matrix S as

$$S = LDL^*$$

where L is a lower triangular square matrix with unity diagonal elements, D is a diagonal matrix, and L^* is the Hermitian (complex conjugate) transpose of L. Only the diagonal and lower triangle of the input matrix are used, and any imaginary component of the diagonal entries is disregarded.

The block's output is a composite matrix with lower triangle elements l_{ij} from L, diagonal elements d_{ij} from D, and upper triangle elements u_{ij} from L^*. It is always sample based. The output format is shown below for a 5-by-5 matrix.

$$u_{ij} = l_{ji}^*$$

LDL factorization requires half the computation of Gaussian elimination (LU decomposition), and is always stable. It is more efficient that Cholesky factorization because it avoids computing the square roots of the diagonal elements.

The algorithm requires that the input be square and Hermitian positive definite. When the input is not positive definite, the block reacts with the behavior specified by the **Non-positive definite input** parameter.

The following options are available:

- Ignore — Proceed with the computation and *do not* issue an alert. The output is *not* a valid factorization. A partial factorization will be present in the upper left corner of the output.

- Warning — Display a warning message in the MATLAB Command Window, and continue the simulation. The output is *not* a valid factorization. A partial factorization will be present in the upper left corner of the output.

- Error — Display an error dialog box and terminate the simulation.

Note The **Non-positive definite input** parameter is a diagnostic parameter. Like all diagnostic parameters on the Configuration Parameters dialog box, it is set to Ignore in the Real-Time Workshop code generated for this block.

Examples   LDL decomposition of a 3-by-3 Hermitian positive definite matrix:

LDL Factorization

**Dialog
Box**

Block Parameters: LDL Factorization dialog box showing:

LDL Factorization (mask)

Computes unit lower triangular L and diagonal D such that S=LDL' for square, symmetric/Hermitian, positive definite input matrix S. Uses only the lower triangle of S.

Parameters

Non-positive definite input: Warning

[OK] Cancel Help Apply

Non-positive definite input
Response to nonpositive definite matrix inputs.

References   Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

**Supported
Data
Types**
- Double-precision floating point
- Single-precision floating point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Cholesky Factorization	Signal Processing Blockset
LDL Inverse	Signal Processing Blockset
LDL Solver	Signal Processing Blockset
LU Factorization	Signal Processing Blockset
QR Factorization	Signal Processing Blockset

See "Factoring Matrices" on page 6-9 for related information.

Purpose   Compute inverse of Hermitian positive definite matrix using LDL factorization

Library   Math Functions / Matrices and Linear Algebra / Matrix Inverses
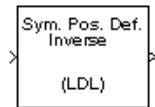
Description   The LDL Inverse block computes the inverse of the Hermitian positive definite input matrix S by performing an LDL factorization.

```
Sym. Pos. Def.
   Inverse
    (LDL)
```

$$S^{-1} = (LDL^*)^{-1}$$

L is a lower triangular square matrix with unity diagonal elements, D is a diagonal matrix, and L^* is the Hermitian (complex conjugate) transpose of L. Only the diagonal and lower triangle of the input matrix are used, and any imaginary component of the diagonal entries is disregarded. The output is always sample based.

LDL factorization requires half the computation of Gaussian elimination (LU decomposition), and is always stable. It is more efficient than Cholesky factorization because it avoids computing the square roots of the diagonal elements.

The algorithm requires that the input be Hermitian positive definite. When the input is not positive definite, the block reacts with the behavior specified by the **Non-positive definite input** parameter. The following options are available:

- Ignore — Proceed with the computation and *do not* issue an alert. The output is *not* a valid inverse.

- Warning — Display a warning message in the MATLAB command window, and continue the simulation. The output is *not* a valid inverse.

- Error — Display an error dialog box and terminate the simulation.

LDL Inverse

Note The **Non-positive definite input** parameter is a diagnostic parameter. Like all diagnostic parameters on the Configuration Parameters dialog box, it is set to Ignore in the Real-Time Workshop code generated for this block.

Dialog Box

Non-positive definite input
> Response to nonpositive definite matrix inputs.

References   Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

Supported Data Types

- Double-precision floating point
- Single-precision floating point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Cholesky Inverse	Signal Processing Blockset
LDL Factorization	Signal Processing Blockset
LDL Solver	Signal Processing Blockset

LU Inverse	Signal Processing Blockset
Pseudoinverse	Signal Processing Blockset
inv	MATLAB

See "Inverting Matrices" on page 6-10 for related information.

LDL Solver

Purpose Solve *SX=B* for *X* when *S* is square Hermitian positive definite matrix

Library Math Functions / Matrices and Linear Algebra / Linear System Solvers
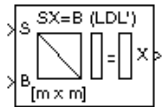
Description The LDL Solver block solves the linear system *SX=B* by applying LDL factorization to the matrix at the S port, which must be square (*M*-by-*M*) and Hermitian positive definite. Only the diagonal and lower triangle of the matrix are used, and any imaginary component of the diagonal entries is disregarded. The input to the B port is the right side *M*-by-*N* matrix, *B*. The output is the unique solution of the equations, *M*-by-*N* matrix *X*, and is always sample based.

A length-*M* 1-D vector input for right side *B* is treated as an *M*-by-1 matrix.

When the input is not positive definite, the block reacts with the behavior specified by the **Non-positive definite input** parameter. The following options are available:

- Ignore — Proceed with the computation and *do not* issue an alert. The output is *not* a valid solution.

- Warning — Proceed with the computation and display a warning message in the MATLAB Command Window. The output is *not* a valid solution.

- Error — Display an error dialog box and terminate the simulation.

Note The **Non-positive definite input** parameter is a diagnostic parameter. Like all diagnostic parameters on the Configuration Parameters dialog box, it is set to Ignore in the Real-Time Workshop code generated for this block.

Algorithm The LDL algorithm uniquely factors the Hermitian positive definite input matrix S as

$$S = LDL^*$$

where *L* is a lower triangular square matrix with unity diagonal elements, *D* is a diagonal matrix, and L^* is the Hermitian (complex conjugate) transpose of *L*.

The equation

$$LDL^*X = B$$

is solved for *X* by the following steps:

1 Substitute

$$Y = DL^*X$$

2 Substitute

$$Z = L^*X$$

3 Solve one diagonal and two triangular systems.

$$LY = B$$

$$DZ = Y$$

$$L^*X = Z$$

Dialog Box
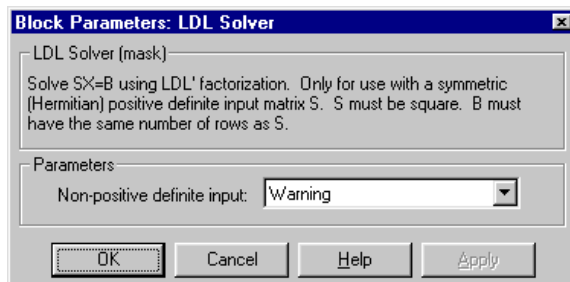
LDL Solver

Non-positive definite input
Response to nonpositive definite matrix inputs.

Supported Data Types

- Double-precision floating point
- Single-precision floating point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Autocorrelation LPC	Signal Processing Blockset
Cholesky Solver	Signal Processing Blockset
LDL Factorization	Signal Processing Blockset
LDL Inverse	Signal Processing Blockset
Levinson-Durbin	Signal Processing Blockset
LU Solver	Signal Processing Blockset
QR Solver	Signal Processing Blockset

See "Solving Linear Systems" on page 6-7 for related information.

Purpose   Design and implement a least-squares FIR filter.

Library   dspobslib

Description
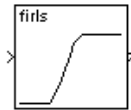
Note The Least Squares FIR Filter Design block is still supported but is likely to be obsoleted in a future release. We strongly recommend replacing this block with the Digital Filter block.

The Least Squares FIR Filter Design block designs an FIR filter and applies it to a discrete-time input using the Direct Form II Transpose Filter block. The filter design uses the `firls` function in the Signal Processing Toolbox to minimize the integral of the squared error between the desired frequency response and the actual frequency response.

An *M*-by-*N* sample-based matrix input is treated as *M**N* independent channels, and an *M*-by-*N* frame-based matrix input is treated as *N* independent channels. In both cases, the block filters each channel independently over time, and the output has the same size and frame status as the input.

The **Filter type** parameter allows you to specify one of the following filters:

- Multiband — The Multiband filter designs a linear-phase filter with an arbitrary magnitude response.

- Differentiator — The Differentiator filter approximates the ideal differentiator. Differentiators are antisymmetric FIR filters with approximately linear magnitude responses. To obtain the

correct derivative, scale the **Gains at these frequencies** vector by π F_s rad/s, where F_s is the sample frequency in Hertz.

- `Hilbert Transformer` — The `Hilbert Transformer` filter approximates the ideal Hilbert transformer. Hilbert transformers are antisymmetric FIR filters with approximately constant magnitude.

The **Band-edge frequency vector** parameter is a vector of frequency points in the range `0` to `1`, where `1` corresponds to half the sample frequency. This vector must have even length, and intermediate points must appear in ascending order. The **Gains at these frequencies** parameter is a vector containing the desired magnitude response at the corresponding points in the **Band-edge frequency vector**.

Each odd-indexed frequency-amplitude pair defines the left endpoint of a line segment representing the desired magnitude response in that frequency band. The corresponding even-indexed frequency-amplitude pair defines the right endpoint. Between the frequency bands specified by these end-points, there may be undefined sections of the specified frequency response. These are called "don't care" or "transition" regions, and the magnitude response in these areas is a result of the optimization in the other (specified) frequency ranges.

Band edge frequency = $\quad [0 \; 0.3 \; 0.4 \; 0.6 \; 0.7 \quad 1]$

Gains = $\quad\quad\quad\quad [0 \quad 1 \quad 0 \quad 0 \; 0.5 \; 0.5]$

Band:

The **Weights** parameter is a vector that specifies the emphasis to be placed on minimizing the error in certain frequency bands relative to others. This vector specifies one weight per band, so it is half the length of the **Band-edge frequency vector** and **Gains at these frequencies** vectors.

In most cases, differentiators and Hilbert transformers have only a single band, so the weight is a scalar value that does not affect the final filter. However, the **Weights** parameter is useful when using the block to design an antisymmetric multiband filter, such as a Hilbert transformer with stopbands.

For more information on the **Band-edge frequency vector**, **Gains at these frequencies**, and **Weights** parameters, see "Filter Designs and Implementation" in the Signal Processing Toolbox documentation. For more on the FIR filter algorithm, see the description of the `firls` function in the Signal Processing Toolbox documentation.

Least Squares FIR Filter Design

Examples

Example 1: Multiband

Consider a lowpass filter with a transition band in the normalized frequency range 0.4 to 0.5, and 10 times more error minimization in the stopband than the passband. In this case,

- **Filter type** = Multiband
- **Band-edge frequency vector** = [0 0.4 0.5 1]
- **Gains at these frequencies** = [1 1 0 0]
- **Weights** = [1 10]

Example 2: Differentiator

Assume the specifications for a differentiator filter require it to have order 21. The "ramp" response extends over the entire frequency range. In this case, specify:

- **Filter type** = Differentiator
- **Filter order** = 21
- **Band-edge frequency vector** = [0 1]
- **Gains at these frequencies** = [0 pi*Fs]

For a type III (even order) filter, the differentiation band should stop short of half the sample frequency. For example, if the filter order is 20, you could specify the block parameters as follows:

- **Filter type** = Differentiator
- **Filter order** = 20
- **Band-edge frequency vector** = [0 0.9]
- **Gains at these frequencies** = [0 0.9*pi*Fs]

Example 3: Hilbert Transformer

Assume the specifications for a Hilbert transformer filter require it to have order 21. The passband extends over approximately the entire frequency range. In this case, specify:

- **Filter type** = Hilbert Transform
- **Filter order** = 21
- **Band-edge frequency vector** = [0.1 1]
- **Gains at these frequencies** = [1 1]

Dialog Box
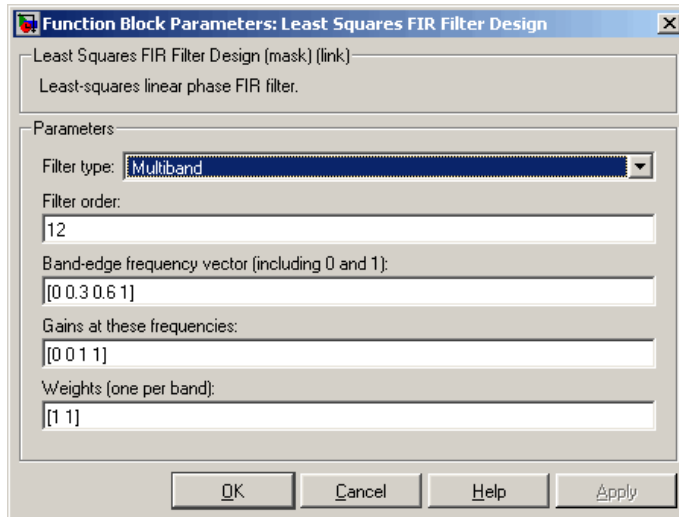
Filter type

    The filter type. Tunable.

Filter order

    The filter order.

Least Squares FIR Filter Design

Band-edge frequency vector

A vector of frequency points, in ascending order, in the range 0 to 1. The value 1 corresponds to half the sample frequency. This vector must have even length. Tunable.

Gains at these frequencies

A vector of frequency-response amplitudes corresponding to the points in the **Band-edge frequency vector**. This vector must be the same length as the **Band-edge frequency vector**. Tunable.

Weights

A vector containing one weight for each frequency band. This vector must be half the length of the **Band-edge frequency vector** and **Gains at these frequencies** vectors. Tunable.

References

Oppenheim, A. V. and R. W. Schafer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989.

Proakis, J. and D. Manolakis. *Digital Signal Processing*. 3rd ed. Englewood Cliffs, NJ: Prentice-Hall, 1996.

Purpose   Compute polynomial coefficients that best fit input data in least squares sense

Library   Math Functions / Polynomial Functions

Description   The Least Squares Polynomial Fit block computes the coefficients of the nth order polynomial that best fits the input data in the least squares sense, where you specify n in the **Polynomial order** parameter. A distinct set of $n+1$ coefficients is computed for each column of the M-by-N input, u.

For a given input column, the block computes the set of coefficients, c_1, c_2, ..., c_{n+1}, that minimizes the quantity

$$\sum_{i=1}^{M} (u_i - \hat{u}_i)^2$$

where u_i is the ith element in the input column, and

$$\hat{u}_i = f(x_i) = c_1 x_i^n + c_2 x_i^{n-1} + \cdots + c_{n+1}$$

The values of the independent variable, x_1, x_2, ..., x_M, are specified as a length-M vector by the **Control points** parameter. The same M control points are used for all N polynomial fits, and can be equally or unequally spaced. The equivalent MATLAB code is shown below.

```
c = polyfit(x,u,n)      % Equivalent MATLAB code
```

Inputs can be frame based or sample based. For convenience, a length-M 1-D vector input is treated as an M-by-1 matrix.

Each column of the $(n+1)$-by-N output matrix, c, represents a set of $n+1$ coefficients describing the best-fit polynomial for the corresponding column of the input. The coefficients in each column are arranged in order of descending exponents, c_1, c_2, ..., c_{n+1}. The output is always sample based.

Least Squares Polynomial Fit

Examples   In the model below, the Polynomial Evaluation block uses the second-order polynomial

$$y = -2u^2 + 3$$

to generate four values of dependent variable *y* from four values of independent variable *u*, received at the top port. The polynomial coefficients are supplied in the vector [-2 0 3] at the bottom port. Note that the coefficient of the first-order term is zero.
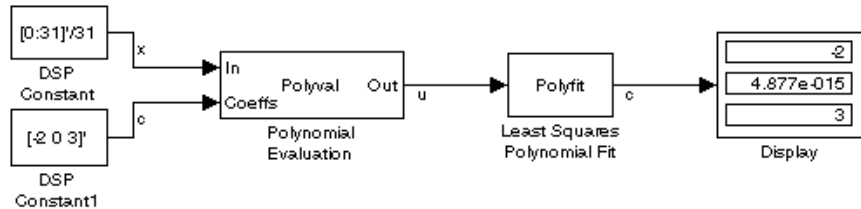
The **Control points** parameter of the Least Squares Polynomial Fit block is configured with the same four values of independent variable *u* that are used as input to the Polynomial Evaluation block, [1 2 3 4]. The Least Squares Polynomial Fit block uses these values together with the input values of dependent variable *y* to reconstruct the original polynomial coefficients.

Dialog Box
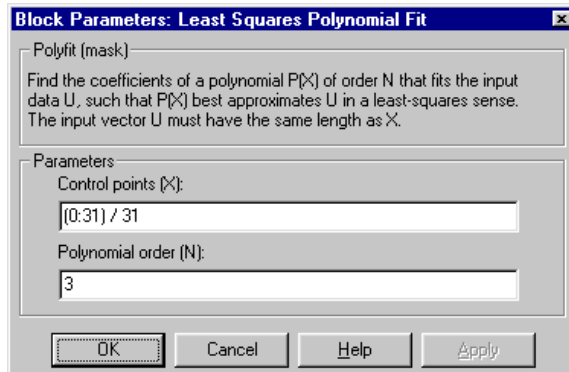
Control points

The values of the independent variable to which the data in each input column correspond. For an *M*-by-*N* input, this parameter must be a length-*M* vector. Tunable.

Polynomial order

The order, n, of the polynomial to be used in constructing the best fit. The number of coefficients is $n+1$. Nontunable.

Supported Data Types

- Double-precision floating point
- Single-precision floating point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Detrend	Signal Processing Blockset
Polynomial Evaluation	Signal Processing Blockset
Polynomial Stability Test	Signal Processing Blockset
polyfit	MATLAB

Levinson-Durbin

Purpose   Solve linear system of equations using Levinson-Durbin recursion

Library   Math Functions / Matrices and Linear Algebra / Linear System Solvers

Description   The Levinson-Durbin block solves the nth-order system of linear equations

$$Ra = b$$

for the particular case where R is a Hermitian, positive-definite, Toeplitz matrix and b is identical to the first column of R shifted by one element and with the opposite sign.
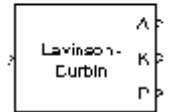
$$\begin{bmatrix} r(1) & r^*(2) & \cdots & r^*(n) \\ r(2) & r(1) & \cdots & r^*(n-1) \\ \vdots & \vdots & \ddots & \vdots \\ r(n) & r(n-1) & \cdots & r(1) \end{bmatrix} \begin{bmatrix} a(2) \\ a(3) \\ \vdots \\ a(n+1) \end{bmatrix} = \begin{bmatrix} -r(2) \\ -r(3) \\ \vdots \\ -r(n+1) \end{bmatrix}$$

The input to the block, r = [$r(1)$ $r(2)$ ... $r(n+1)$], can be a 1-D or 2-D vector (row or column). It contains lags 0 through n of an autocorrelation sequence, which appear in the matrix R.

The block can output the polynomial coefficients, A, the reflection coefficients, K, and the prediction error power, P, in various combinations. The **Output(s)** parameter allows you to enable the A and K outputs by selecting one of the following settings:

- A — Port A outputs A=[1 $a(2)$ $a(3)$ ... $a(n+1)$], the solution to the Levinson-Durbin equation. A has the same dimension as the input. The elements of the output can also be viewed as the coefficients of an nth-order autoregressive (AR) process (see below).

- K — Port K outputs K=[$k(1)$ $k(2)$ ... $k(n)$], which contains n reflection coefficients, and has the same dimension as the input, less one element. (A scalar input causes an error when you select K.) Reflection coefficients are useful for realizing a lattice representation of the AR process described below.

- A and K — The block outputs both representations at their respective ports. (A scalar input causes an error when you select A and K.)

Both *A* and *K* are always 1-D vectors.

The prediction error power, *P*, (a scalar), is output when you select the **Output prediction error power (P)** check box. *P* represents the power of the output of an FIR filter with taps *A* and input autocorrelation described by *r*, where *A* represents a prediction error filter and *r* is the input to the block. (In this case, *A* is a whitening filter.)

When you select the **If the value of lag 0 is zero, A=[1 zeros], K=[zeros], P=0** check box (default), an input whose $r(1)$ element is zero generates a zero-valued output. When you do *not* select this check box, an input with $r(1) = 0$ generates NaNs in the output. In general, an input with $r(1) = 0$ is invalid because it does not construct a positive-definite matrix *R*; however, it is common for blocks to receive zero-valued inputs at the start of a simulation. The check box allows you to avoid propagating NaNs during this period.

Applications

One application of the Levinson-Durbin formulation above is in the Yule-Walker AR problem, which concerns modeling an unknown system as an autoregressive process. Such a process would be modeled as the output of an all-pole IIR filter with white Gaussian noise input. In the Yule-Walker problem, the use of the signal's autocorrelation sequence to obtain an optimal estimate leads to an $Ra = b$ equation of the type shown above, which is most efficiently solved by Levinson-Durbin recursion. In this case, the input to the block represents the autocorrelation sequence, with $r(1)$ being the zero-lag value. The output at the block's A port then contains the coefficients of the autoregressive process that optimally models the system. The coefficients are ordered in descending powers of *z*, and the AR process is minimum phase. The prediction error, *G*, defines the gain for the unknown system, where $G = \sqrt{P}$.

$$H(z) = \frac{G}{A(z)} = \frac{G}{1 + a(2)z^{-1} + \ldots + a(n+1)z^{-n}}$$

The output at the block's K port contains the corresponding reflection coefficients, $[k(1)\ k(2)\ \ldots\ k(n)]$, for the lattice realization of this IIR filter. The Yule-Walker AR Estimator block implements this autocorrelation-based method for AR model estimation, while the Yule-Walker Method block extends the method to spectral estimation.

Another common application of the Levinson-Durbin algorithm is in linear predictive coding, which is concerned with finding the coefficients of a moving average (MA) process (or FIR filter) that predicts the next value of a signal from the current signal sample and a finite number of past samples. In this case, the input to the block represents the signal's autocorrelation sequence, with $r(1)$ being the zero-lag value, and the output at the block's A port contains the coefficients of the predictive MA process (in descending powers of z).

$$H(z) = A(z) = 1 + a(2)z^{-1} + \ldots + a(n+1)z^{-n}$$

These coefficients solve the optimization problem below.

$$\min_{\{a_i\}} E\left[\left|x_n - \sum_{i=1}^{N} a_i x_{n-i}\right|^2\right]$$

Again, the output at the block's K port contains the corresponding reflection coefficients, $[k(1)\ k(2)\ \ldots\ k(n)]$, for the lattice realization of this FIR filter. The Autocorrelation LPC block in the Linear Prediction library implements this autocorrelation-based prediction method.

Fixed-Point Data Types

The diagrams in this section show the data types used within the Levinson-Durbin block for fixed-point signals.

After initialization, n updates are performed. At the (j+1) update,

$$\text{value in accumulator} = r(j+1) + \sum a_j(i) \times r(j-i+1)$$

The diagram below displays the fixed-point data types used in this calculation:
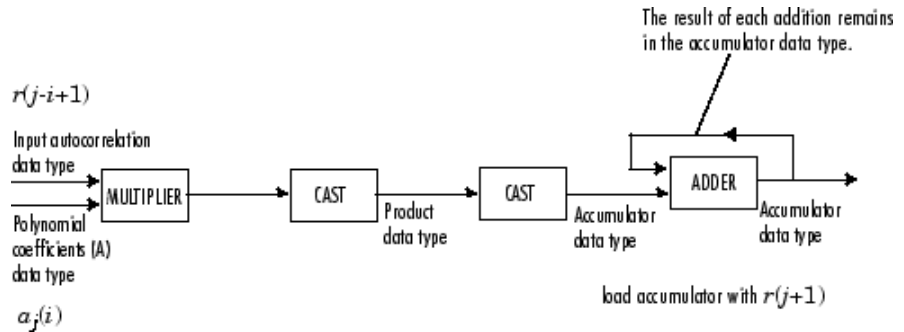
The reflection coefficients K are then updated according to

$$K_{j+1} = \text{ value in accumulator}/P_j$$

The prediction error power P is then updated according to

$$P_{j+1} = P_j - P_j \times K_{j+1} \times \text{conj}(K_{j+1})$$

The diagram below displays the fixed-point data types used in this calculation:
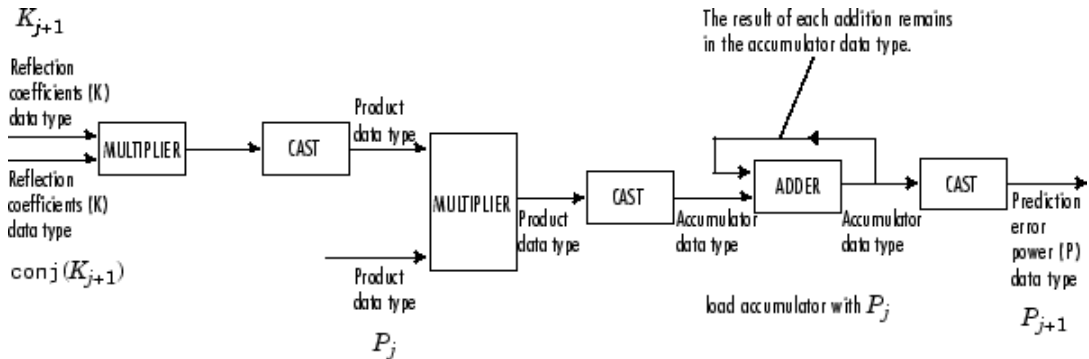
Levinson-Durbin

The polynomial coefficients A are then updated according to

$$a_{j+1}(i) = a_j(i) + K_{j+1} \times \mathrm{conj}(a_j(j-1+i))$$

The diagram below displays the fixed-point data types used in this calculation:
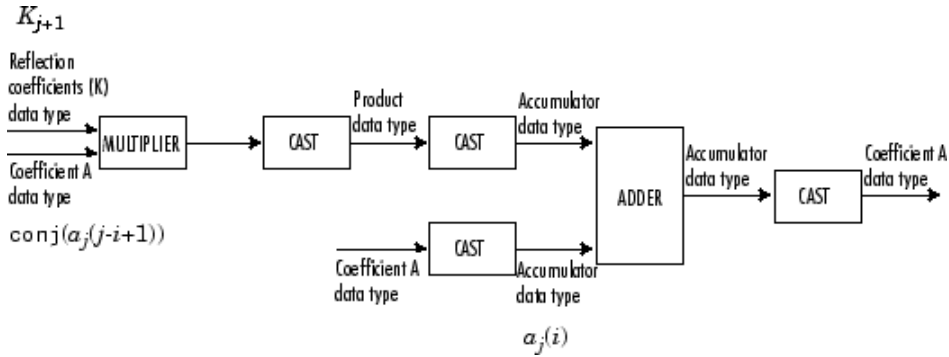
Algorithm   The algorithm requires $O(n^2)$ operations, and is thus much more efficient for large n than standard Gaussian elimination, which requires $O(n^3)$ operations.

Dialog Box

The **Main** pane of the Levinson-Durbin block dialog appears as follows:
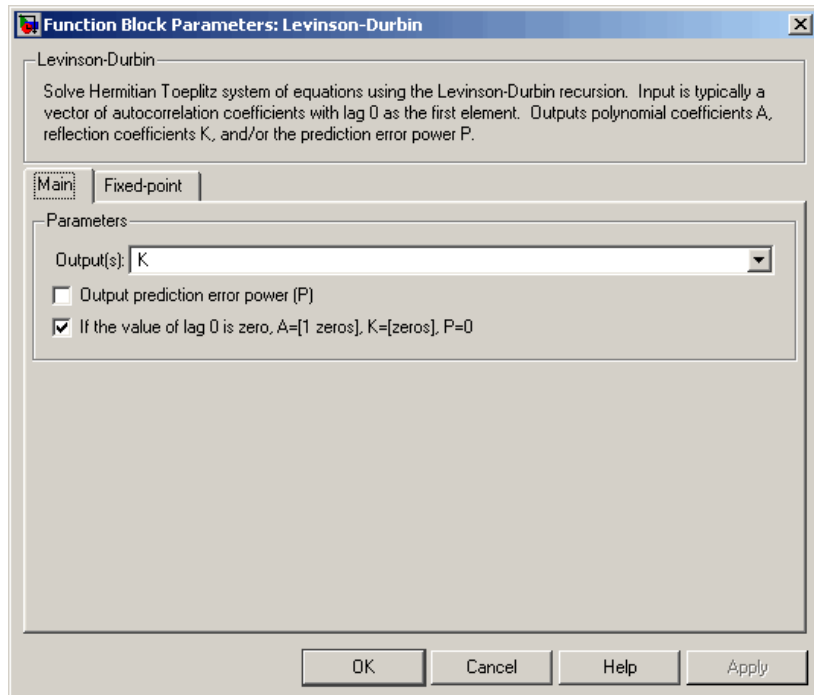
Output(s)

Specify the solution representation of Ra = b to output: model coefficients (A), reflection coefficients (K), or both (A and K). For scalar inputs, this parameter must be set to A.

Output prediction error power (P)

Select to output the prediction error at port P.

If the value of lag 0 is zero, A=[1 zeros], K=[zeros], P=0

Set to output a zero-vector for inputs having r(1) = 0. Otherwise, the block outputs NaNs for these inputs.

The **Fixed-point** pane of the Levinson-Durbin block dialog appears as follows:
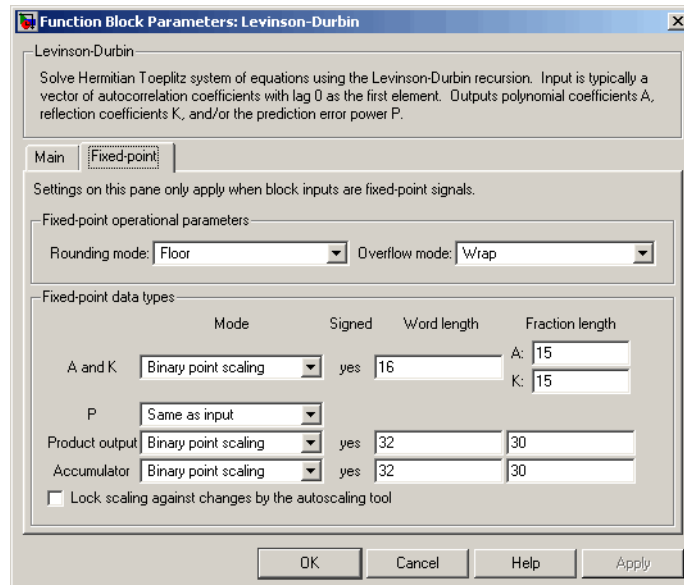
Rounding mode

> Select the rounding mode for fixed-point operations.

Overflow mode

> Select the overflow mode for fixed-point operations.

A

> Use this parameter to designate how you would like to specify the word and fraction lengths of the polynomial coefficients (*A*). Refer to "Fixed-Point Data Types" on page 10-596 for illustrations depicting the use of the polynomial coefficients data type in this block.
>
> • When you select Binary point scaling, you are able to enter the word length and fraction length of *A*, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of *A*. This block requires power-of-two slope and a bias of zero.

K

Use this parameter to designate how you would like to specify the word and fraction lengths of the reflection coefficients (*K*). Refer to "Fixed-Point Data Types" on page 10-596 for illustrations depicting the use of the reflection coefficients data type in this block.

- When you select `Binary point scaling`, you are able to enter the word length and fraction length of *K*, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of *K*. This block requires power-of-two slope and a bias of zero.

P

Use this parameter to designate how you would like to specify the word and fraction lengths of the prediction error power (*P*). Refer to "Fixed-Point Data Types" on page 10-596 for illustrations depicting the use of the prediction error power data type in this block.

- When you select `Same as input`, these characteristics will match those of the input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and fraction length of *P*, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of *P*. This block requires power-of-two slope and a bias of zero.

Product output

Use this parameter to designate how you would like to specify the product output word and fraction lengths. Refer to "Fixed-Point Data Types" on page 10-596 for illustrations depicting the use of the product output data type in this block.

- When you select `Same as input`, these characteristics will match those of the input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and fraction length of the product output, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the product output. This block requires power-of-two slope and a bias of zero.

Accumulator

Use this parameter to designate how you would like to specify the accumulator word and fraction lengths. Refer to "Fixed-Point Data Types" on page 10-596 for illustrations depicting the use of the accumulator data type in this block.

- When you select `Same as product output`, these characteristics will match those of the product output.

- When you select `Same as input`, these characteristics will match those of the input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and fraction length of the accumulator, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the accumulator. This block requires power-of-two slope and a bias of zero.

References

Golub, G. H., and C. F. Van Loan. Sect. 4.7 in *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

Ljung, L. *System Identification: Theory for the User*. Englewood Cliffs, NJ: Prentice Hall, 1987. Pgs. 278-280.

Kay, Steven M., *Modern Spectral Estimation: Theory and Application*. Englewood Cliffs, NJ: Prentice Hall, 1988.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point (signed only)
- 8-, 16-, and 32-bit signed integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Cholesky Solver	Signal Processing Blockset
LDL Solver	Signal Processing Blockset
Autocorrelation LPC	Signal Processing Blockset
LU Solver	Signal Processing Blockset
QR Solver	Signal Processing Blockset
Yule-Walker AR Estimator	Signal Processing Blockset
Yule-Walker Method	Signal Processing Blockset
levinson	Signal Processing Toolbox

See "Solving Linear Systems" on page 6-7 for related information.

LMS Adaptive Filter

Purpose   Compute filter estimates for an input using the LMS adaptive filter algorithm

Library   Filtering / Adaptive Filters

Description
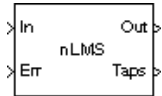
> **Note** The LMS Adaptive Filter block is still supported but is likely to be obsoleted in a future release. We recommend replacing this block with the LMS Filter block.

The LMS Adaptive Filter block implements an adaptive FIR filter using the stochastic gradient algorithm known as the normalized least mean-square (LMS) algorithm.

$$y(n) = \hat{w}^H(n-1)u(n)$$
$$e(n) = d(n) - y(n)$$
$$\hat{w}(n) = \hat{w}(n-1) + \frac{u(n)}{a + u^H(n)u(n)}\mu e^*(n)$$

The variables are as follows.

Variable	Description
n	The current algorithm iteration
$u(n)$	The buffered input samples at step n
$\hat{w}(n)$	The vector of filter-tap estimates at step n
$y(n)$	The filtered output at step n
$e(n)$	The estimation error at step n
$d(n)$	The desired response at step n
μ	The adaptation step size

To overcome potential numerical instability in the tap-weight update, a small positive constant (a = 1e-10) has been added in the denominator.

To turn off normalization, clear the **Use normalization** check box in the parameter dialog box. The block then computes the filter-tap estimate as

$$\hat{w}(n) = \hat{w}(n-1) + u(n)\mu e^*(n)$$

The block icon has port labels corresponding to the inputs and outputs of the LMS algorithm. Note that inputs to the In and Err ports must be sample-based scalars. The signal at the Out port is a scalar, while the signal at the Taps port is a sample-based vector.

Block Ports	Corresponding Variables
In	u, the scalar input, which is internally buffered into the vector $u(n)$
Out	$y(n)$, the filtered scalar output
Err	$e(n)$, the scalar estimation error
Taps	$\hat{w}(n)$, the vector of filter-tap estimates

An optional Adapt input port is added when you select the **Adapt input** check box in the dialog box. When this port is enabled, the block continuously adapts the filter coefficients while the Adapt input is nonzero. A zero-valued input to the Adapt port causes the block to stop adapting, and to hold the filter coefficients at their current values until the next nonzero Adapt input.

The **FIR filter length** parameter specifies the length of the filter that the LMS algorithm estimates. The **Step size** parameter corresponds to µ in the equations. Typically, for convergence in the mean square, µ must be greater than 0 and less than 2. The **Initial value of filter taps** specifies the initial value $\hat{w}(0)$ as a vector, or as a scalar to be repeated for all vector elements. The **Leakage factor** specifies the

LMS Adaptive Filter

value of the leakage factor, $1 - \mu\alpha$, in the leaky LMS algorithm below. This parameter must be between 0 and 1.

$$\hat{w}(n+1) = (1 - \mu\alpha)\hat{w}(n) + \frac{u(n)}{u^H(n)u(n)}\mu e^*(n)$$

Examples   See the `lmsadeq`, `lmsadlp`, and `lmsadtde` demos.

Dialog Box
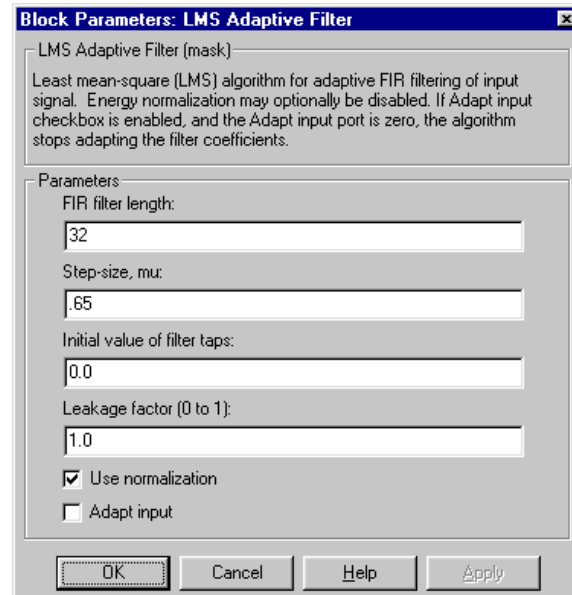
FIR filter length
>   The length of the FIR filter.

Step-size
>   The step-size, usually in the range (0, 2). Tunable.

Initial value of filter taps
>   The initial FIR filter coefficients.

Leakage factor

The leakage factor, in the range [0, 1]. Tunable.

Use normalization

Select this check box to compute the filter-tap estimate using the normalized equations.

Adapt input

Enables the Adapt port when selected.

References   Haykin, S. *Adaptive Filter Theory*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1996.

Supported Data Types

- Double-precision floating point

- Single-precision floating point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Kalman Adaptive Filter	Signal Processing Blockset
RLS Adaptive Filter	Signal Processing Blockset

See "Adaptive Filters" on page 3-53 for related information.

LMS Filter

Purpose   Compute filtered output, filter error, and filter weights for a given input and desired signal using LMS adaptive filter algorithm

Library   Filtering / Adaptive Filters

Description
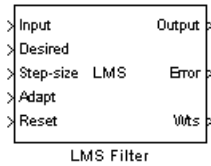
The LMS Filter block can implement an adaptive FIR filter using five different algorithms. The block estimates the filter weights, or coefficients, needed to minimize the error, $e(n)$, between the output signal, $y(n)$, and the desired signal, $d(n)$. Connect the signal you want to filter to the Input port. This input signal can be a sample-based scalar or a single-channel frame-based signal. Connect the desired signal to the Desired port. The desired signal must have the same data type, frame status, complexity, and dimensions as the input signal. The Output port outputs the filtered input signal, which is the estimate of the desired signal. The output of the Output port has the same frame status as the input signal. The Error port outputs the result of subtracting the output signal from the desired signal.

When you select LMS for the **Algorithm** parameter, the block calculates the filter weights using the least mean-square (LMS) algorithm. This algorithm is defined by the following equations.

$$
\begin{aligned}
y(n) &= \mathbf{w}^T(n-1)\mathbf{u}(n) \\
e(n) &= d(n) - y(n) \\
\mathbf{w}(n) &= \mathbf{w}(n-1) + f(\mathbf{u}(n), e(n), \mu)
\end{aligned}
$$

The weight update function, for the LMS adaptive filter algorithm, is defined as

$$
f(\mathbf{u}(n), e(n), \mu) = \mu e(n)\mathbf{u}^*(n)
$$

The variables are as follows.

Variable	Description
n	The current time index
$\mathbf{u}(n)$	The vector of buffered input samples at step n

Variable	Description
$\mathbf{u}^*(n)$	The complex conjugate of the vector of buffered input samples at step n
$\mathbf{w}(n)$	The vector of filter weight estimates at step n
$y(n)$	The filtered output at step n
$e(n)$	The estimation error at step n
$d(n)$	The desired response at step n
μ	The adaptation step size

When you select Normalized LMS for the **Algorithm** parameter, the block calculates the filter weights using the normalized LMS algorithm. The weight update function, for the normalized LMS algorithm, is defined as

$$f(\mathbf{u}(n), e(n), \mu) \;=\; \mu e(n) \frac{\mathbf{u}^*(n)}{a + \mathbf{u}^H(n)\mathbf{u}(n)}$$

To overcome potential numerical instability in the update of the weights, a small positive constant, epsilon, has been added in the denominator. For double-precision floating point input, epsilon is 2.2204460492503131e-016. For single-precision floating point input, epsilon is 1.192092896e-07.

When you select Sign-Error LMS for the **Algorithm** parameter, the block calculates the filter weights using the LMS algorithm equations. However, each time the block updates the weights, it replaces the error term, $e(n)$, with +1 when the error term is positive or -1 when the error term is negative.

When you select Sign-Data LMS for the **Algorithm** parameter, the block calculates the filter weights using the LMS algorithm equations. However, each time the block updates the weights, it replaces each sample of the input vector, $\mathbf{u}(n)$, with +1 when the input sample is positive or -1 is the input sample is negative.

When you select `Sign-Sign LMS` for the **Algorithm** parameter, the block calculates the filter weights using the LMS algorithm equations. However, each time the block updates the weights, it replaces the error term, $e(n)$, with +1 when the error term is positive or -1 is the error term is negative. It also replaces each sample of the input vector, $\mathbf{u}(n)$, with +1 when the input sample is positive or -1 is the input sample is negative.

Use the **Filter length** parameter to specify the length of the filter weights vector.

The **Step size (mu)** parameter corresponds to µ in the equations. For convergence of the normalized LMS equations, 0<µ<2. You can either specify a step size using the input port, Step-size, or by entering a value in the Block Parameters: LMS Filter dialog box.

Use the **Leakage factor (0 to 1)** parameter to specify the leakage factor, $1 - \mu\alpha$, where $0 < 1 - \mu\alpha \leq 1$, in the leaky LMS algorithm shown below.

$$\mathbf{w}(n) = (1 - \mu\alpha)\mathbf{w}(n-1) + f(\mathbf{u}(n), e(n), \mu)$$

When you select `LMS` from the **Algorithm** list, the weight update function in the above equation is the LMS weight update function. When you select `Normalized LMS` from the **Algorithm** list, the weight update function in the above equation is the normalized LMS weight update function.

Enter the initial filter weights, $\mathbf{w}(0)$, as a vector or a scalar in the **Initial value of filter weights** text box. When you enter a scalar, the block uses the scalar value to create a vector of filter weights. This vector has length equal to the filter length and all of its values are equal to the scalar value.

When you select the **Adapt port** check box, an Adapt port appears on the block. When the input to this port is greater than zero, the block continuously updates the filter weights. When the input to this port is less than or equal to zero, the filter weights remain at their current values.

When you want to reset the value of the filter weights to their initial values, use the **Reset port** parameter. The block resets the filter weights whenever a reset event is detected at the Reset port. The reset signal rate must be the same rate as the data signal input.

From the **Reset port** list, select None to disable the Reset port. To enable the Reset port, select one of the following from the **Reset port** list:

- Rising edge — Triggers a reset operation when the Reset input does one of the following:

  - Rises from a negative value to a positive value or zero

  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)

- Falling edge — Triggers a reset operation when the Reset input does one of the following:

  - Falls from a positive value to a negative value or zero

  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero (see the following figure)

- `Either edge` — Triggers a reset operation when the Reset input is a `Rising edge` or `Falling edge` (as described above)

- `Non-zero sample` — Triggers a reset operation at each sample time that the Reset input is not zero

---

Note When running simulations in the Simulink `MultiTasking` mode, sample-based reset signals have a one-sample latency, and frame-based reset signals have one frame of latency. Thus, there is a one-sample or one-frame delay between the time the block detects a reset event, and when it applies the reset. For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" on page 2-56 and "Models with Multiple Sample Rates" in the Real-Time Workshop User's Guide documentation.

Select the **Output filter weights** check box to create a Wts port on the block. For each iteration, the block outputs the current updated filter weights from this port.

Fixed-Point Data Types

The following diagrams show the data types used within the LMS Filter block for fixed-point signals; the table summarizes the definitions of variables used in the diagrams:

Variable	Definition
X (**u** in the LMS Filter equations)	Input vector
W	Vector of filter weights
mu	Step size
e	Error
Q	Quotient, $$Q = \frac{mu * e}{X'X}$$
Product **X'X**	Product data type in Energy calculation diagram
Accumulator **X'X**	Accumulator data type in Energy calculation diagram
Product **W'X**	Product data type in Convolution diagram
Accumulator **W'X**	Accumulator data type in Convolution diagram
Product mu*e	Product data type in Product of step size and error diagram
Product Q***X**	Product and accumulator data type in Weight update diagram. [a]

a. The accumulator data type for this quantity is automatically set to be the same as the product data type. The minimum, maximum, and overflow information for this accumulator is logged as part of the product information. Autoscaling treats this product and accumulator as one data type.

Energy calculation (for normalized LMS algorithm only)

Input data type → MULTIPLIER → Product (X'X) data type → CAST → Accumulator (X'X) data type → ADDER → Accumulator (X'X) data type

Convolution

Input data type / Weight data type → MULTIPLIER → Product (W'X) data type → CAST → Accumulator (W'X) data type → ADDER → Accumulator (W'X) data type → CAST → Output data type (same as input signal data type)

Output error signal

Desired signal data type / Output signal data type → SUBTRACTOR → Error signal data type
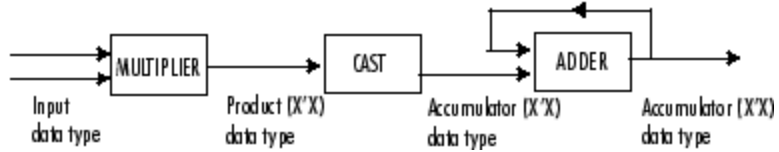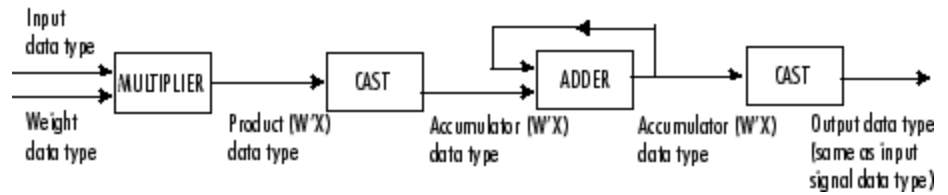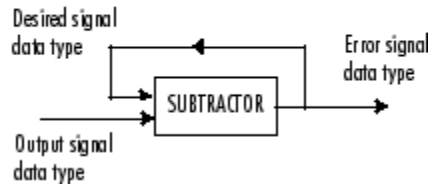
Product of step size and error (for LMS and Sign-Data LMS algorithms only)

Step size data type / Error data type → MULTIPLIER → Product (mu*e) data type
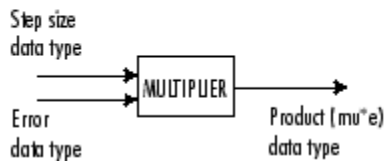
Quotient (for normalized LMS only)
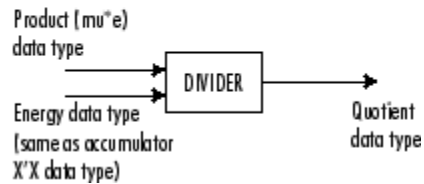
Weight update
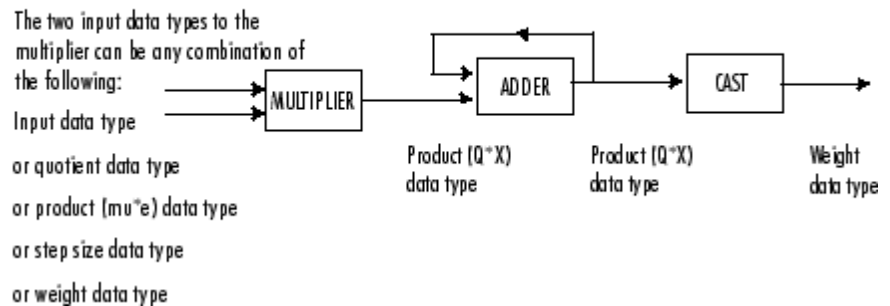
You can set the data type of the parameters, weights, products, quotient, and accumulators in the block mask.

Fixed-point inputs, outputs, and mask parameters of this block must have the following characteristics:

- The input signal and the desired signal must have the same word length, but their fraction lengths can differ.

- The step size and leakage factor must have the same word length, but their fraction lengths can differ.

- The output signal and the error signal have the same word length and the same fraction length as the desired signal.

- The quotient and the product output of the **X'X**, **W'X**, mu*e, and Q***X** operations must have the same word length, but their fraction lengths can differ.

• The accumulator data type of the **X'X** and **W'X** operations must have the same word length, but their fraction lengths can differ.

The output of the multiplier is in the product output data type if at least one of the inputs to the multiplier is real. If both of the inputs to the multiplier are complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed, refer to "Multiplication Data Types" on page 8-16.

Dialog Box

The **Main** pane of the LMS Filter block dialog appears as follows:
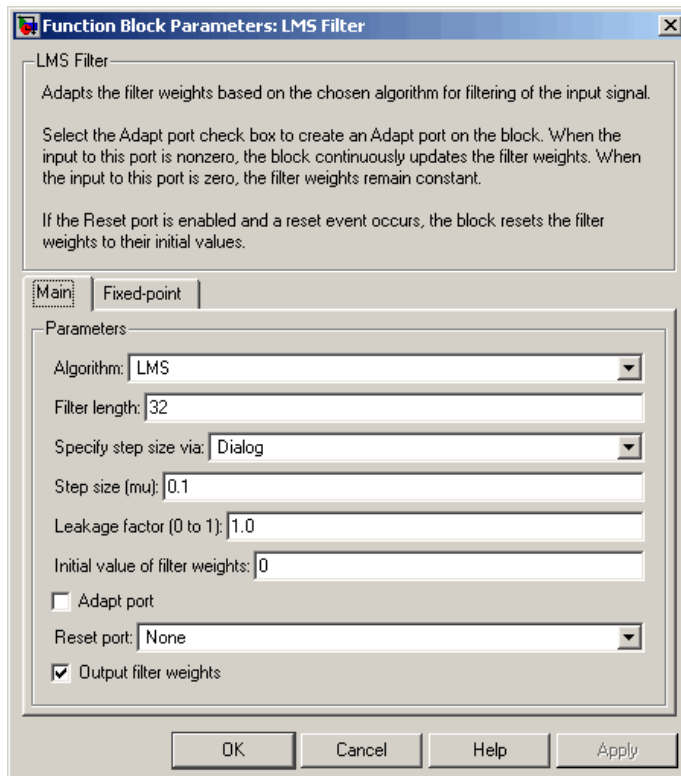
Algorithm

Choose the algorithm used to calculate the filter weights.

Filter length

Enter the length of the FIR filter weights vector.

Specify step size via

Select `Dialog` to enter a value for step size in the Block parameters: LMS Filter dialog box. Select `Input port` to specify step size using the Step-size input port.

Step size (mu)

Enter the step size. Tunable.

Leakage factor (0 to 1)

Enter the leakage factor, $0 < 1 - \mu\alpha \leq 1$. Tunable.

Initial value of filter weights

Specify the initial values of the FIR filter weights.

Adapt port

Select this check box to enable the Adapt input port.

Reset port

Select this check box to enable the Reset input port.

Output filter weights

Select this check box to export the filter weights from the Wts port.

The **Fixed-point** pane of the LMS Filter block dialog appears as follows:

LMS Filter
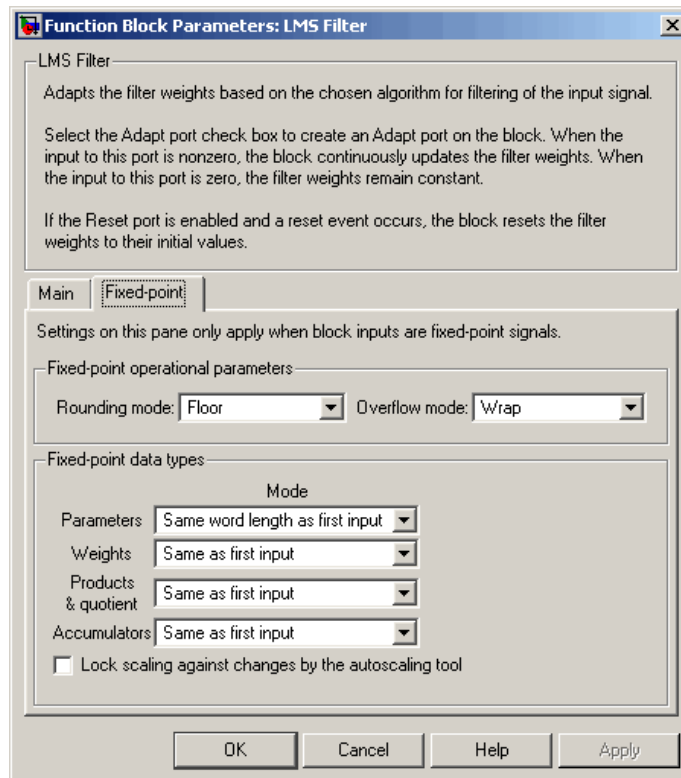
Rounding mode

    Select the rounding mode for fixed-point operations.

Overflow mode

    Select the overflow mode for fixed-point operations.

Parameters

    This parameter is visible if, for the **Specify step size via** parameter, you choose Dialog. Choose how you will specify the word length and the fraction length of the leakage factor and step size:

- When you select `Same word length as first input`, the word length of the leakage factor and step size will match that of the first input to the block. In this mode, the fraction length of the leakage factor and step size is automatically set to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.

- When you select `Specify word length`, you are able to enter the word length of the leakage factor and step size, in bits. In this mode, the fraction length of the leakage factor and step size is automatically set to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the leakage factor and step size, in bits. The leakage factor and the step size must have the same word length, but the fraction lengths can differ.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the leakage factor and step size. The leakage factor and the step size must have the same word length, but the slopes can differ. This block requires a power-of-two slope and a bias of zero.

If, for the **Specify step size via** parameter, you choose `Input port`, the word length of the leakage factor is the same as the word length of the step size input at the Step size port. The fraction length of the leakage factor is automatically set to the best precision possible based on the word length of the leakage factor.

Weights

Choose how you will specify the word length and fraction length of the filter weights of the block:

- When you select `Same as first input`, the word length and fraction length of the filter weights will match those of the first input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the filter weights, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the filter weights. This block requires a power-of-two slope and a bias of zero.

Products & quotient

Choose how you will specify the word length and fraction length of **X'X**, **W'X**, mu*e, Q***X**, and the quotient, Q. Here, **X** is the input vector, which is **u** in the LMS filter equations. **W** is the vector of filter weights, mu is the step size, e is the error, and Q is the quotient, which is defined as

$$Q = \frac{mu^*e}{X'X}$$

- When you select `Same as first input`, the word length and fraction length of these quantities will match those of the first input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of these quantities, in bits. The word length of the quantities must be the same, but the fraction lengths can differ.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of these quantities. The word length of the quantities must be the same, but the slopes can differ. This block requires a power-of-two slope and a bias of zero.

Accumulators

Use this parameter to specify how you would like to designate the word and fraction lengths of the accumulators for the **X'X** and **W'X** operations.

> **Note** This parameter is *not* used to designate the word and fraction lengths of the accumulator for the $Q*\mathbf{X}$ operation. The accumulator data type for this quantity is automatically set to be the same as the product data type. The minimum, maximum, and overflow information for this accumulator is logged as part of the product information. Autoscaling treats this product and accumulator as one data type.

Refer to "Fixed-Point Data Types" on page 10-612 and "Multiplication Data Types" on page 8-16 for illustrations depicting the use of the accumulator data type in this block:

- When you select `Same as first input`, these characteristics will match those of the input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the accumulators, in bits. The word length of both the accumulators must be the same, but the fraction lengths can differ.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the accumulators. The word length of both the accumulators must be the same, but the slopes can differ. This block requires a power-of-two slope and a bias of zero.

References   Hayes, M.H. *Statistical Digital Signal Processing and Modeling*. New York: John Wiley & Sons, 1996.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point
Desired	• Double-precision floating point • Single-precision floating point • Fixed point
Step-size	• Double-precision floating point • Single-precision floating point • Fixed point
Adapt	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Reset	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point

Port	Supported Data Types
Error	• Double-precision floating point • Single-precision floating point • Fixed point
Wts	• Double-precision floating point • Single-precision floating point • Fixed point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Kalman Adaptive Filter	Signal Processing Blockset
RLS Filter	Signal Processing Blockset
Block LMS Filter	Signal Processing Blockset
Fast Block LMS Filter	Signal Processing Blockset

See "Adaptive Filters" on page 3-53 for related information.

LPC to LSF/LSP Conversion

Purpose   Convert linear prediction coefficients (LPCs) to line spectral pairs (LSPs) or line spectral frequencies (LSFs)

Library   Estimation / Linear Prediction

Description   The LPC to LSF/LSP Conversion block takes a vector of linear prediction coefficients (LPCs) and converts it to a vector of line spectral pairs (LSPs) or line spectral frequencies (LSFs). When converting LPCs to LSFs, the block outputs match those of the `poly2lsf` function.

The input LPCs, 1, a_1, a_2, ... a_m, must be the denominator of the transfer function of a stable all-pole filter with the form given in the first equation of "Requirements for Valid Outputs" on page 10-624. A length-*M*+1 input yields a length-*M* output. Inputs can be sample- or frame-based vectors, but outputs are always sample-based vectors.

See other sections of this reference page to learn about how to ensure that you get valid outputs, how to detect invalid outputs, how the block computes the LSF/LSP values, and more.

Requirements for Valid Outputs

To get *valid outputs*, your inputs and the **Root finding coarse grid points** parameter value must meet these requirements:

- The input LPCs, 1, a_1, a_2, ... a_m, must come from the denominator of the following transfer function, *H(z),* of a stable all-pole filter (all roots of *H(z)* must be inside the unit circle). Note that the first term in *H(z)*'s denominator must be 1. When the input LPCs do not come from a transfer function of the following form, the block outputs are invalid.

$$H(z) = \frac{1}{1 + a_1 z^{-1} + a_2 z^{-2} + \dots + a_m z^{-m}}$$

- The **Root finding coarse grid points** parameter value must be large enough so that the block can find all the LSP or LSF values.

(The output LSFs and LSPs are roots of polynomials related to the input LPC polynomial; the block looks for these roots to produce the output. For details, see "LSF and LSP Computation Method: Chebyshev Polynomial Method for Root Finding" on page 10-634.) When you do not set **Root finding coarse grid points** to a high enough value relative to the number of LPCs, the block might not find all the LSPs or LSFs and yield invalid outputs as described in "Root Finding Method Limitations: Failure to Find Roots" on page 10-636.

To learn about recognizing invalid inputs and outputs and parameters for dealing with them, see "Handling and Recognizing Invalid Inputs and Outputs" on page 10-628.

Setting Outputs to LSFs or LSPs

Set the **Output** parameter to one of the following settings to determine whether the block outputs LSFs or LSPs:

- LSF in radians (0 pi) — Block outputs the LSF values between 0 and π radians in increasing order. The block does not output the guaranteed LSF values, 0 and π.

- LSF normalized in range (0 0.5) — Block outputs *normalized* LSF values in increasing order, computed by dividing the LSF values between 0 and π radians by 2π. The block does not output the guaranteed normalized LSF values, 0 and 0.5.

- LSP in range (-1 1) — Block outputs LSP values in decreasing order, equal to the cosine of the LSF values between 0 and π radians. The block does not output the guaranteed LSP values, -1 and 1.

Adjusting Output Computation Time and Accuracy with Root Finding Parameters

The values n and k determine the block's output computation time and accuracy, where

- n is the value of the **Root finding coarse grid points** parameter (choose this value with care; see the note below)

- k is the value of the **Root finding bisection refinement** parameter.

- Decreasing the values of n and k decreases the output computation time, but also decreases output accuracy:

  - The upper bound of block's computation time is proportional to $k \cdot (n-1)$.

  - Each LSP output is within $1/(n \cdot 2^k)$ of the actual LSP value.

  - Each LSF output is within ΔLSF of the actual LSF value, LSF_{act}, where

$$\Delta LSF = \left| \mathrm{acos}(LSF_{act}) - \mathrm{acos}(LSF_{act} + 1/(n \cdot 2^k)) \right|$$

Note When the value of the **Root finding coarse grid points** parameter is too small relative to the number of LPCs, the block might output *invalid data* as described in "Requirements for Valid Outputs" on page 10-624. Also see "Handling and Recognizing Invalid Inputs and Outputs" on page 10-628.

Valid Inputs and Corresponding Outputs

The following list and table summarize characteristics of valid inputs and the corresponding outputs.

Notable Input and Output Properties

- To get valid outputs, your input LPCs and the value of the **Root finding coarse grid points** parameter must meet the requirements described in "Requirements for Valid Outputs" on page 10-624.

- Block treats each column of an input matrix as a set of LPCs

- Length-L+1 input yields length-L output

- Output is always sample based

- **Output** parameter determines the output type (see "Setting Outputs to LSFs or LSPs" on page 10-625):

    - LSFs — frequencies, w_k, where $0 < w_k < \pi$ and $w_k < w_{k+1}$

    - Normalized LSFs — $w_k / 2\pi$

    - LSPs — $\cos(w_k)$

Input and Output Dimensions, Sizes, and Frame Statuses

Valid LPC Input	LSF and LSP Outputs(Always Sample-Based)
Sample-based length-M+1 row vector, $M > 0$ $\begin{bmatrix} 1 & a_1 & a_2 & \cdots & a_m \end{bmatrix}$ *Frame-based row vectors are not valid inputs.*	Sample-based length-M row vector LSF in radians: $\begin{bmatrix} w_1 & w_2 & \cdots & w_m \end{bmatrix}$ LSF normalized: $\frac{1}{2\pi} \cdot \begin{bmatrix} w_1 & w_2 & \cdots & w_m \end{bmatrix}$ LSP: $\begin{bmatrix} \cos(w_1) & \cos(w_2) & \cdots & \cos(w_m) \end{bmatrix}$
Sample- or frame-based length-M+1 column vector, $M \geq 0$ $\begin{bmatrix} 1 \\ a_1 \\ a_2 \\ \vdots \\ a_m \end{bmatrix}$	Sample-based length-M column vector LSF in radians: $\begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{bmatrix}$ LSF normalized: $\frac{1}{2\pi} \cdot \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{bmatrix}$ LSP: $\begin{bmatrix} \cos(w_1) \\ \cos(w_2) \\ \vdots \\ \cos(w_m) \end{bmatrix}$
1-D length-M+1 unoriented vector, $M > 0$ $(1, a_1, a_2, \ldots, a_m)$	1-D length-M unoriented vector LSF in radians: $(w_1, w_2, \ldots, w_m)$ LSF normalized: $\frac{1}{2\pi} \cdot (w_1, w_2, \ldots, w_m)$ LSP: $(\cos(w_1), \cos(w_2), \ldots, \cos(w_m))$

Handling and Recognizing Invalid Inputs and Outputs

The block outputs invalid data when your input LPCs and the value of the **Root finding coarse grid points** parameter do not meet the requirements described in "Requirements for Valid Outputs" on page 10-624. The following topics describe what invalid outputs look like, and how to set the block parameters provided for handling invalid inputs and outputs:

- "What Invalid Outputs Look Like" on page 10-628
- "Parameters for Handling Invalid Inputs and Outputs" on page 10-629

What Invalid Outputs Look Like

Invalid outputs have the same dimensions, sizes, and frame statuses as valid outputs, which you can look up in Input and Output Dimensions, Sizes, and Frame Statuses on page 10-627. However, invalid outputs do not contain all the LSP or LSF values. Instead, invalid outputs contain none or some of the LSP and LSF values and the rest of the output vector or matrix is filled with *place holder values* (-1, 0.5, or π depending on the **Output** parameter setting).

In short, all invalid outputs end in one of the place holder values (-1, 0.5, or π) as illustrated in the following table. To learn how to use the block's parameters for handling invalid inputs and outputs, see the next section.

Output Parameter Setting	Place Holder	Sample Invalid Outputs
LSF in radians (0 pi)	π	$\begin{bmatrix} w_1 & w_2 & w_3 & \pi & \pi & \pi & \pi \end{bmatrix}$

Output Parameter Setting	Place Holder	Sample Invalid Outputs
LSF normalized in range (0 0.5)	0.5	$$\begin{bmatrix} w_1 \\ w_2 \\ 0.5 \end{bmatrix}$$
LSP in range (-1 1)	-1	$$\begin{bmatrix} \cos(w_{13}) \\ \cos(w_{23}) \\ -1 \\ -1 \\ -1 \end{bmatrix}$$

Parameters for Handling Invalid Inputs and Outputs

You must set how the block handles invalid inputs and outputs by setting these parameters:

- **Show output validity status (1=valid, 0=invalid)** — Setting this parameter activates a second block output port that outputs a 1 when the output is valid, and a 0 when they are invalid. The LSF and LSP outputs are *invalid* when the block fails to find all the LSF or LSP values or when the input LPCs are unstable (for details, see "Requirements for Valid Outputs" on page 10-624). See the previous section to learn how to recognize invalid outputs.

- **If current output is invalid, overwrite with previous output** — Selecting this check box causes the block to overwrite invalid outputs with the *previous* output. When you set this parameter you also need to consider these parameters:

  - **When first output is invalid, overwrite with user-defined values** — When the *first* input is unstable, you can choose to either overwrite the invalid first output with the default values (by *clearing* this parameter) or with values you specify (by *selecting* this check box and specifying the values in the parameter

described next). The default initial overwrite values are the LSF or LSP representations of an all-pass filter.

- **User-defined LSP/LSF values for overwriting invalid first output** — In this parameter you specify the values for overwriting an invalid first output if you selected the **When first output is invalid, overwrite with user-defined values**. The vector of LSP/LSF values you specify should have the same dimension, size, and frame status as the other outputs, which you can look up in Input and Output Dimensions, Sizes, and Frame Statuses on page 10-627.

- **If first input value is not 1** — The block output is invalid when the first coefficient in an LPC vector is not 1; this parameter determines what the block does when given such inputs:

  - `Ignore` — Proceed with computations as if the first coefficient is 1.

  - `Normalize` — Divide the input LPCs by the value of the first coefficient before computing the output.

  - `Normalize and warn` — In addition to `Normalize`, display a warning message at the MATLAB command line.

  - `Error` — Stop the simulation and display an error message at the MATLAB command line.

Dialog Box
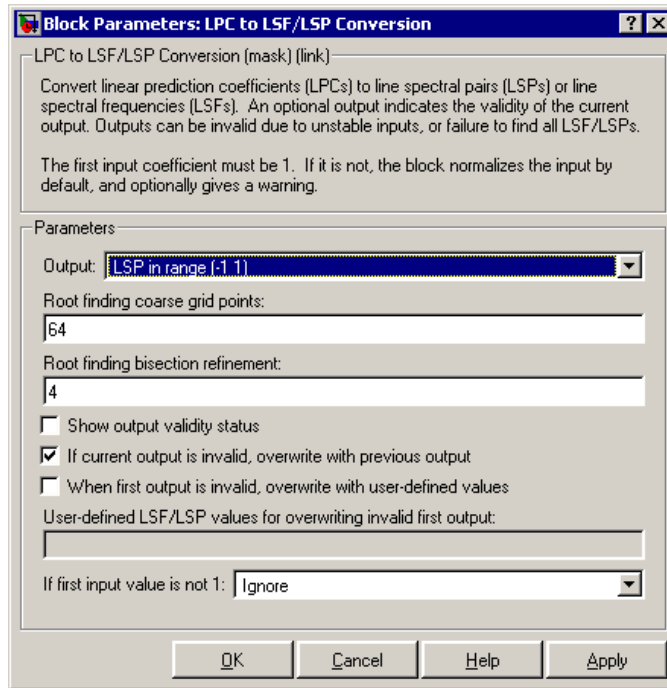
Output

Specifies whether to convert the input linear prediction polynomial coefficients (LPCs) to LSP in range (-1 1), LSF in radians (0 pi), or LSF normalized in range (0 0.5). See "Setting Outputs to LSFs or LSPs" on page 10-625 for descriptions of the three settings.

Root finding coarse grid points

The value n, where the block divides the interval (-1, 1) into n subintervals of equal length, and looks for roots (LSP values) in each subinterval. You must pick n large enough or the block output might be invalid as described in "Requirements for Valid Outputs" on page 10-624. To learn how the block uses this parameter to compute the output, see "LSF and LSP Computation Method: Chebyshev Polynomial Method for Root Finding" on

page 10-634. Also see "Adjusting Output Computation Time and Accuracy with Root Finding Parameters" on page 10-625. Tunable.

Root finding bisection refinement

The value k, where each LSP output is within $1/(n \cdot 2^k)$ of the actual LSP value, where n is the value of the **Root finding coarse grid points** parameter. To learn how the block uses this parameter to compute the output, see "LSF and LSP Computation Method: Chebyshev Polynomial Method for Root Finding" on page 10-634. Also see "Adjusting Output Computation Time and Accuracy with Root Finding Parameters" on page 10-625. Tunable.

Show output validity status

Selecting this check box activates a second block output port that outputs a 1 when the output is valid, and a 0 when they are invalid. For more information, see "Handling and Recognizing Invalid Inputs and Outputs" on page 10-628.

If current output is invalid, overwrite with previous output

Selecting this check box causes the block to overwrite invalid outputs with the *previous* output. Setting this parameter activates other parameters for taking care of initial overwrite values (when the very first output of the block is invalid). For more information, see "Parameters for Handling Invalid Inputs and Outputs" on page 10-629.

When first output is invalid, overwrite with user-defined values

When the *first* input is unstable, you can choose to either overwrite the invalid first output with the default values (by *clearing* this check box) or with values you specify (by *setting* this check box). The default initial overwrite values are the LSF or LSP representations of an all-pass filter. For more information, see "Parameters for Handling Invalid Inputs and Outputs" on page 10-629.

User-defined LSP/LSF values for overwriting invalid first output

In this parameter you specify the values for overwriting an invalid first output when you select **When first output is invalid, overwrite with user-defined values**. The vector or matrix of

LSP/LSF values you specify should have the same dimension, size, and frame status as the other outputs, which you can look up in the table Input and Output Dimensions, Sizes, and Frame Statuses on page 10-627.

If first input value is not 1

Determines what the block does when the first coefficient of an input is not 1. The block can either proceed with computations as when the first coefficient is 1 (Ignore); divide the input LPCs by the value of the first coefficient before computing the output (Normalize); in addition to Normalize, display a warning message at the MATLAB command line (Normalize and warn); stop the simulation and display an error message at the MATLAB command line (Error). For more information, see "Parameters for Handling Invalid Inputs and Outputs" on page 10-629.

Supported Data Types

- Double-precision floating point

- Single-precision floating point

- Boolean — Supported only by the optional output port that appears when you set the parameter, **Show output validity status (1=valid, 0=invalid)**

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

References Kabal, P. and Ramachandran, R. "The Computation of Line Spectral Frequencies Using Chebyshev Polynomials."*IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. ASSP-34 No. 6, December 1986. pp. 1419-1426.

LPC to LSF/LSP Conversion

Theory

LSF and LSP Computation Method: Chebyshev Polynomial Method for Root Finding

---

Note To learn the principles on which the block's LSP and LSF computation method is based, see the reference listed in "References" on page 10-633.

To compute *LSP outputs*, the block relies on the fact that LSP values are the *roots of two particular polynomials* related to the input LPC polynomial; the block finds these roots using the Chebyshev polynomial root finding method, described next. To compute *LSF outputs*, the block computes the arc cosine of the LSPs, outputting values ranging from 0 to π radians.

Root Finding Method

LSPs, which are the *roots of two particular polynomials*, always lie in the range (-1, 1). (The guaranteed roots at 1 and -1 are factored out.) The block finds the LSPs by looking for a sign change of the two polynomials' values between points in the range (-1, 1). The block searches a maximum of $k(n-1)$ points, where

- n is the value of the **Root finding coarse grid points** parameter
- k is the value of the **Root finding bisection refinement** parameter

The block's method for choosing which points to check consists of the following two steps:

1 **Coarse Root Finding** —- The block divides the interval [-1, 1] into n intervals, each of length $2/n$, and checks the signs of both polynomials' values at the endpoints of the intervals. The block starts checking signs at 1, and continues checking signs at $1-4/n$, $1-6/n$, and so on at steps of length $2/n$, outputting any point if it is a root. The block *stops searching* in these situations:

a The block finds a sign change of a polynomial's values between two adjacent points. An interval containing a sign change is guaranteed to contain a root, so the block further searches the interval as described in Step 2, Root Finding Refinement.

b The block finds and outputs all *M* roots (given a length-*M*+1 LPC input).

c The block fails to find all *M* roots and yields invalid outputs as described in "Handling and Recognizing Invalid Inputs and Outputs" on page 10-628.

2 Root Finding Refinement — When the block finds a sign change in an interval, $[a, b]$, it searches for the root guaranteed to lie in the interval by following these steps:

a Check if Midpoint Is a Root — The block checks the sign of the midpoint of the interval $[a, b]$. The block outputs the midpoint if it is a root, and continues Step 1, Coarse Root Finding, at the next point, $a - 2/n$. Otherwise, the block selects the half-interval with endpoints of opposite sign (either $[a, (a + b)/2]$ or $[(a + b)/2, b]$) and executes Step 2b, Stop or Continue Root Finding Refinement.

b Stop or Continue Root Finding Refinement — When the block has repeated Step 2a k times (k is the value of the **Root finding bisection refinement** parameter), the block linearly interpolates the root by using the half-interval's endpoints, outputs the result as an LSP value, and returns to Step 1, Coarse Root Finding. Otherwise, the block repeats Step 2a using the half-interval.

Coarse Root Finding: LSPs are roots of two particular polynomials related to the input LPCs. Check signs of the two polynomials at evenly-spaced points to find all intervals containing a sign change. Output any roots (LSPs) found.

Root finding coarse grid points = 5
Divide [-1, 1] into five intervals of equal length and check signs of the polynomials' values at the endpoints of the intervals: 1, 0.6, 0.2, -0.2, -0.6, -1.
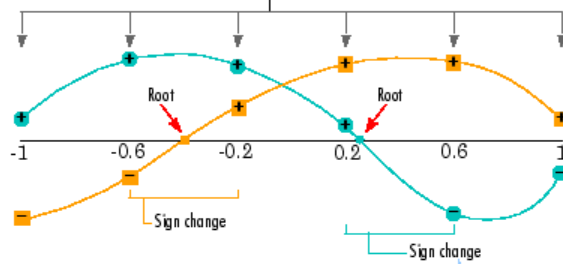
Root Finding Refinement: Whenever Coarse Root Finding identifies an interval containing a sign change, repeatedly bisect the interval to better approximate the root (LSP value).

Bisection 1: Check the sign of the polynomial at the midpoint of the interval and select the half-interval with endpoints of opposite sign: [0.2, 0.4]

Bisection 2: Similar to Bisection 1

Root finding bisection refinement = 3
Bisect all sign change intervals found in the Coarse Root Finding up to three times to find the root. When the root is not found in the last bisection, linearly interpolate the root.

Bisection 3: The last bisection. Since the midpoint of this interval is not the root, linearly interpolate the root and output the result as an LSP value.

Linearly interpolated LSP approximation
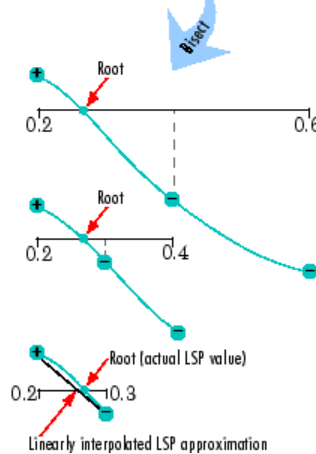
Coarse Root Finding and Root Finding Refinement

Root Finding Method Limitations: Failure to Find Roots

The block root finding method described above can fail, causing the block to produce invalid outputs (for details on invalid outputs, see "Handling and Recognizing Invalid Inputs and Outputs" on page 10-628).

In particular, the block can fail to find some roots if the value of the **Root finding coarse grid points** parameter, n, is too small. If the polynomials oscillate quickly and have roots that are very close

together, the root finding might be too coarse to identify roots that are very close to each other, as illustrated in Fixing a Failed Root Finding on page 10-638.

For higher-order input LPC polynomials, you should increase the **Root finding coarse grid points** value to ensure the block finds all the roots and produces valid outputs.

Root Finding Fails: The root search divides the interval [-1, 1] into four intervals, but all three roots are in a single interval. The block can only find one root per interval, so two of the roots are never found.

Root finding coarse grid points = 4
Divide [-1, 1] into four intervals of equal length and check signs of the polynomials' values at the endpoints of the intervals: 1, 0.5, 0, -0.5, -1.

There are three roots in a single interval

Bisect 1   Missed roots   Bisect 2   Only root found
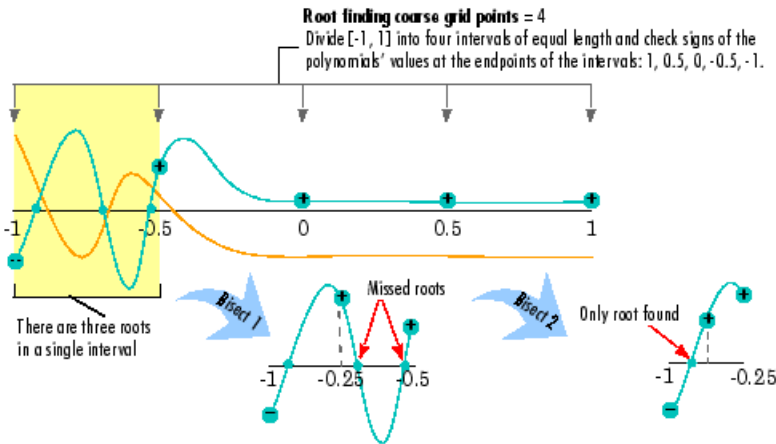
Fix Root Finding so it Succeeds: Increasing the value of the **Root finding coarse grid points** parameter to 15 ensures that each root is in its own interval, so all roots are found.

Root finding coarse grid points = 15

Each root is in its own interval.
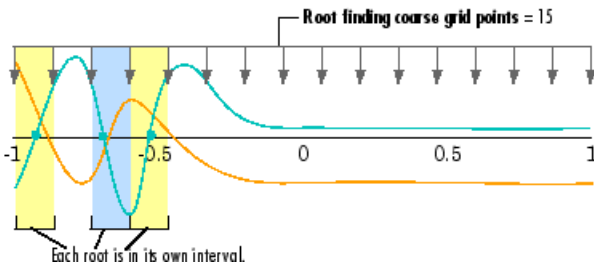
Fixing a Failed Root Finding

See Also

LSF/LSP to LPC Conversion	Signal Processing Blockset
LPC to/from RC	Signal Processing Blockset
LPC/RC to Autocorrelation	Signal Processing Blockset
poly2lsf	Signal Processing Toolbox

Purpose   Convert line spectral frequencies (LSFs) or line spectral pairs (LSPs) to linear prediction coefficients (LPCs)

Library   Estimation / Linear Prediction

Description The LSF/LSP to LPC Conversion block takes a vector or matrix of line spectral pairs (LSPs) or line spectral frequencies (LSFs) and converts it to a vector or matrix of linear prediction polynomial coefficients (LPCs). When converting LSFs to LPCs, the block outputs match those of the lsf2poly function.

The inputs to the block can be in one of three formats that you must indicate in the **Input** parameter, which has the following settings:

- LSF in range (0 pi) — Vector of LSF values between 0 and π radians in increasing order. Do not include the guaranteed LSF values, 0 and π.

- LSF normalized in range (0 0.5) — Vector of *normalized* LSF values in increasing order, (compute by dividing the LSF values between 0 and π radians by 2π). Do not include the guaranteed normalized LSF values, 0 and 0.5.

- LSP in range (-1 1) — Vector of LSP values in decreasing order, equal to the cosine of the LSF values between 0 and π radians. Do not include the guaranteed LSP values, -1 and 1.

Dialog Box
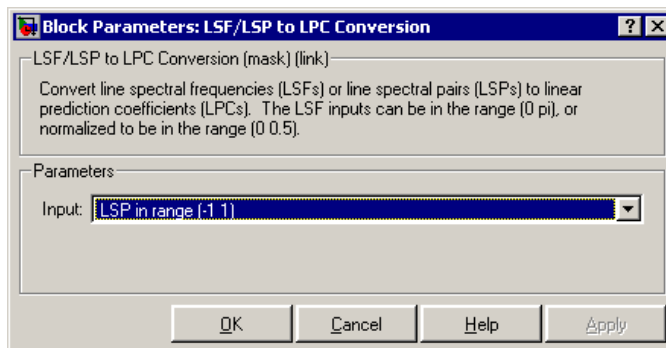
LSF/LSP to LPC Conversion

Input
> Specifies whether to convert LSP in range (-1 1), LSF in range (0 pi), or LSF normalized in range (0 0.5) to linear prediction coefficients (LPCs).

Supported Data Types

- Double-precision floating point
- Single-precision floating point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

References
Kabal, P. and Ramachandran, R. "The Computation of Line Spectral Frequencies Using Chebyshev Polynomials." *IEEE Transactions on Acoustics, Speech, and Signal Processing*, Vol. ASSP-34 No. 6, December 1986. pp. 1419-1426.

See Also

|	
LPC to LSF/LSP Conversion	Signal Processing Blockset
LPC to/from RC	Signal Processing Blockset
LPC/RC to Autocorrelation	Signal Processing Blockset
lsf2poly	Signal Processing Toolbox

Purpose	Convert linear prediction coefficients (LPCs) to cepstral coefficients (CCs) or cepstral coefficients to linear prediction coefficients
Library	Estimation / Linear Prediction

Description
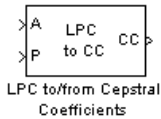
The LPC to/from Cepstral Coefficients block either converts linear prediction coefficients (LPCs) to cepstral coefficients (CCs) or cepstral coefficients to linear prediction coefficients. Set the **Type of conversion** parameter to LPCs to cepstral coefficients or Cepstral coefficients to LPCs to select the domain into which you want to convert your coefficients. The LPC port corresponds to LPCs, and the CC port corresponds to the CCs. For more information, see "Algorithm" on page 10-642.

Consider a signal $x(n)$ as the input to an FIR analysis filter represented by LPCs. The output of this analysis filter, $e(n)$, is known as the prediction error signal. The power of this error signal is denoted by P, the prediction error power.

When you select LPCs to cepstral coefficients from the **Type of conversion** list, you can specify the prediction error power in two ways. From the **Specify P** list, choose via input port to input the prediction error power using input port P. Select assume P equals 1 to set the prediction error power equal to 1.

When you select LPCs to cepstral coefficients from the **Type of conversion** list, the **Output size same as input size** check box appears in the Block Parameters: LPC to/from Cepstral coefficients dialog box. When you select this check box, the length of the input vector of LPCs is equal to the output vector of CCs. When you do not select this check box, enter the length of the output vector of CCs in the **Length of output cepstral coefficients** box. This value must be greater than zero.

When you select LPCs to cepstral coefficients from the **Type of conversion** list, you can use the **If first input value is not 1** parameter to specify the behavior of the block when the first coefficient of the LPC vector is not 1. The following options are available:

LPC to/from Cepstral Coefficients

- `Replace it with 1` —- Changes the first value of the coefficient vector to 1. The other coefficient values are unchanged.

- `Normalize` — Divides the entire vector of coefficients by the first coefficient so that the first coefficient of the LPC vector is 1.

- `Normalize and Warn` — Divides the entire vector of coefficients by the first coefficient so that the first coefficient of the LPC vector is 1. The block displays a warning message telling you that your vector of coefficients has been normalized.

- `Error` — Displays an error telling you that the first coefficient of the LPC vector is not 1.

When you select `Cepstral coefficients to LPCs` from the **Type of conversion** list, the **Output P** check box appears on the block. Select this check box when you want to output the prediction error power from output port P.

Algorithm   The cepstral coefficients are the coefficients of the Fourier transform representation of the logarithm magnitude spectrum. Consider a sequence, $x(n)$, having a Fourier transform $X(\omega)$. The cepstrum, $c_x(n)$, is defined by the inverse Fourier transform of $C_x(\omega)$, where $C_x(\omega) = \log_e X(\omega)$. See the Real Cepstrum block reference page for information on computing cepstrum coefficients from time-domain signals.

LPC to CC

When in this mode, this block uses a recursion technique to convert LPCs to CCs. The LPC vector is defined by $\begin{bmatrix} a_0 & a_1 & a_2 & ... & a_p \end{bmatrix}$ and the CC vector is defined by $\begin{bmatrix} c_0 & c_1 & c_2 & ... & c_p & ... & c_{n-1} \end{bmatrix}$. The recursion is defined by the following equations:

$$c_0 = \log_e E^2$$

$$c_m = a_m + \frac{1}{m} \sum_{k=1}^{m-1} \Big[-(m-k) \cdot a_k \cdot c_{(m-k)} \Big], 1 \le m \le p$$

$$c_m = \sum_{k=1}^{m-1} \left[\frac{-(m-k)}{m} \cdot a_k \cdot c_{(m-k)} \right], p < m < n$$

CC to LPC

When in this mode, this block uses a recursion technique to convert CCs to LPCs. The CC vector is defined by $\begin{bmatrix} c_0 & c_1 & c_2 & ... & c_p & ... & c_n \end{bmatrix}$ and the LPC vector is defined by $\begin{bmatrix} a_0 & a_1 & a_2 & ... & a_p \end{bmatrix}$. The recursion is defined by the following equations

$$a_m = c_m - \frac{1}{m} \sum_{k=1}^{m-1} \Big[(m-k) \cdot c_{(m-k)} \cdot a_k \Big]$$

$$P = \exp(C_0)$$

where $m = 1, 2, ..., p$.

LPC to/from Cepstral Coefficients

Dialog Box
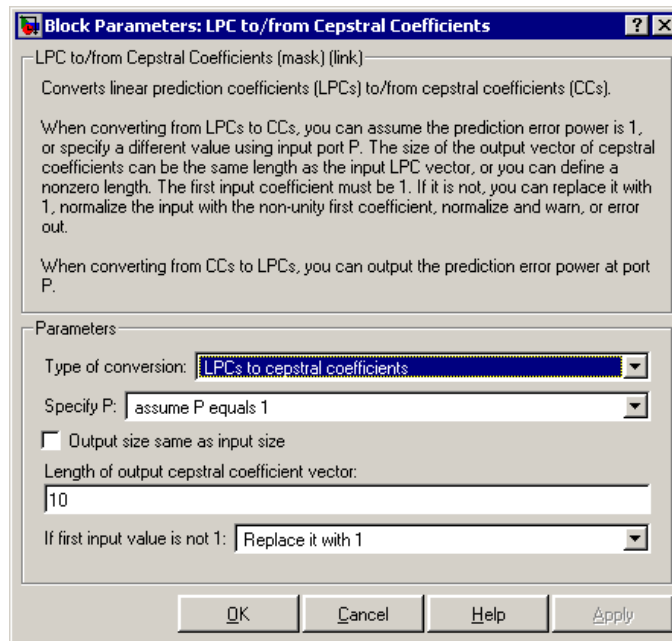
Type of conversion
Choose `LPCs to cepstral coefficients` or `Cepstral coefficients to LPCs` to specify the domain into which you want to convert your coefficients.

Specify P
Choose `via input port` to input the values of prediction error power using input port P. Select `assume P equals 1` to set the prediction error power equal to 1.

Output size same as input size
When you select this check box, the length of the input vector of LPCs is equal to the output vector of CCs.

Length of output cepstral coefficients
Enter the length of the output vector of CCs.

If first input value is not 1
> Select what you would like the block to do when the first coefficient of the LPC vector is not 1. You can choose `Replace it with 1`, `Normalize`, `Normalize and Warn`, and `Error`.

Output P
> Select this check box to output the prediction error power from output port P.

References Rabiner, L and Biing-Hwang Juang, *Fundamentals of Speech Recognition.* Englewood Cliffs, NJ: Prentice Hall, 1993.

Supported Data Types
- Double-precision floating point
- Single-precision floating point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Levinson-Durbin	Signal Processing Blockset
LPC to LSF/LSP Conversion	Signal Processing Blockset
LSF/LSP to LPC Conversion	Signal Processing Blockset
LPC to/from RC	Signal Processing Blockset
LPC/RC to Autocorrelation	Signal Processing Blockset
Real Cepstrum	Signal Processing Blockset
Complex Cepstrum	Signal Processing Blockset

LPC to/from RC

Purpose   Convert linear prediction coefficients (LPCs) to reflection coefficients (RCs) or reflection coefficients to linear prediction coefficients

Library   Estimation / Linear Prediction

Description The LPC to/from RC block either converts linear prediction coefficients (LPCs) to reflection coefficients (RCs) or reflection coefficients to linear prediction coefficients. Set the **Type of conversion** parameter to `LPC to RC` or `RC to LPC` to select the domain into which you want to convert your coefficients. The `A` port corresponds to LPC coefficients, and the `K` port corresponds to the RC coefficients. For more information, see "Algorithm" on page 10-647.

Consider a signal $x(n)$ as the input to an FIR analysis filter represented by LPC coefficients. The output of the this analysis filter, $e(n)$, is known as the prediction error signal. The power of this error signal is denoted by *P*. When the zero lag autocorrelation coefficient of $x(n)$ is one, the autocorrelation sequence and prediction error power are said to be normalized.

Select the **Output normalized prediction error power** check box to enable port `P`. The normalized prediction error power, a scalar, is output at port `P` and varies between zero and one.

Select the **Output LPC filter stability** check box to output the stability of the filter represented by the LPCs or RCs. The synthesis filter represented by the LPCs is stable when the absolute value of each of the roots of the LPC polynomial is less than one. The lattice filter represented by the RCs is stable when the absolute value of each reflection coefficient is less than 1. When the filter is stable, the block outputs a Boolean value of 1 at the `S` port. When the filter is unstable, the block outputs a Boolean value of 0 at the `S` port.

If first input value is not 1 parameter specifies the behavior of the block when the first coefficient of the LPC coefficient vector is not 1. The following options are available:

- `Replace it with 1` — Changes the first value of the coefficient vector to 1. The other coefficient values are unchanged.

- `Normalize` — Divides the entire vector of coefficients by the first coefficient so that the first coefficient of the LPC coefficient vector is 1.

- `Normalize and Warn` — Divides the entire vector of coefficients by the first coefficient so that the first coefficient of the LPC coefficient vector is 1. The block displays a warning message telling you that your vector of coefficients has been normalized.

- `Error` — Displays an error telling you that the first coefficient of the LPC coefficient vector is not 1.

Algorithm   **LPC to RC**

When in this mode, this block uses backward Levinson recursion to convert linear prediction coefficients (LPCs) to reflection coefficients

(RCs). For a given Nth order LPC vector $LPC_N = \begin{bmatrix} 1 & a_{N1} & a_{N2} & \cdots & a_{NN} \end{bmatrix}$, the block calculates the Nth reflection coefficient value using the formula $\gamma_N = -a_{NN}$. The block then finds the lower order LPC vectors , $LPC_{N-1}, LPC_{N-2}, \ldots, LPC_1$, using the following recursion.

for $p = N, N-1, \ldots, 2$,

$$\gamma_p = a_{pp}$$

$$F = 1 - \gamma_p^{\,2}$$

$$a_{p-1, m} = \frac{a_{p, m}}{F} - \frac{\gamma_p a_{p, p-m}}{F}, \, 1 \le m < p$$

end

Finally, $\gamma_1 = -a_{11}$. The reflection coefficient vector is $\begin{bmatrix} \gamma_1, \gamma_2, \ldots, \gamma_N \end{bmatrix}$.

RC to LPC

When in this mode, this block uses Levinson recursion to convert reflection coefficients (RCs) to linear prediction coefficients (LPCs). In

this case, the input to the block is $RC = \begin{bmatrix} \gamma_1 & \gamma_2 & \cdots & \gamma_N \end{bmatrix}$. The zeroth order

LPC vector term is 1. Starting with this term, the block uses recursion to calculate the higher order LPC vectors, $LPC_2, LPC_3, ..., LPC_N$, until it has calculated the entire LPC matrix.

$$LPC_{matrix} = \begin{bmatrix} LPC_0 \\ LPC_1 \\ LPC_2 \\ ... \\ LPC_N \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & ... & 0 \\ 1 & a_{11} & 0 & 0 & ... & 0 \\ 1 & a_{21} & a_{22} & 0 & ... & 0 \\ 1 & a_{31} & a_{32} & a_{33} & ... & 0 \\ ... & ... & ... & ... & ... & ... \\ 1 & a_{N1} & a_{N2} & a_{N3} & ... & a_{NN} \end{bmatrix}$$

This LPC matrix consists of LPC vectors of order 0 through N found by using the Levinson recursion. The following are the formulas for the recursion steps, for $p = 0, 1, ..., N-1$.

$$a_{p+1,m} = a_{p,m} + \gamma_{p+1} a_{p,p+1-m}, 1 \le m \le p$$

$$a_{p+1,p+1} = \gamma_{p+1}$$

Dialog Box
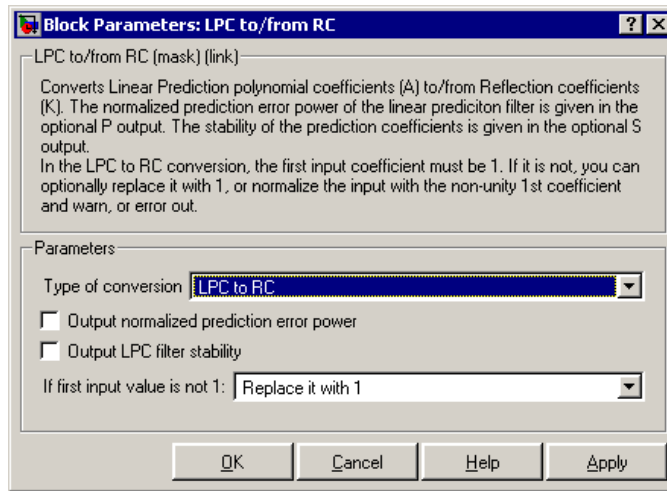
Type of conversion

Select LPC to RC or RC to LPC to select the domain into which you want to convert your coefficients.

Output normalized prediction error power

Select this check box to output the normalized prediction error power at port P.

Output LPC filter stability

Select this check box to output the stability of the filter. When the filter represented by the LPCs or RCs is stable, the block outputs a Boolean value of 1 at the S port. When the filter represented by the LPCs or RCs is unstable, the block outputs a Boolean value of 0 at the S port.

If first input value is not 1

Select what you would like the block to do when the first coefficient of the LPC coefficient vector is not 1. You can choose Replace it with 1, Normalize, Normalize and Warn, and Error.

References   Makhoul, J *Linear Prediction: A tutorial review.* Proc. IEEE. 63, 63, 56 (1975).

Markel, J.D. and A. H. Gray, Jr., *Linear Prediction of Speech.* New York, Springer-Verlag, 1976.

Supported Data Types

- Double-precision floating-point
- Single-precision floating-point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Levinson-Durbin	Signal Processing Blockset
LPC to LSF/LSP Conversion	Signal Processing Blockset
LSF/LSP to LPC Conversion	Signal Processing Blockset
LPC/RC to Autocorrelation	Signal Processing Blockset

Purpose   Convert linear prediction coefficients (LPCs) or reflection coefficients (RCs) to autocorrelation coefficients (ACs)

Library   Estimation / Linear Prediction
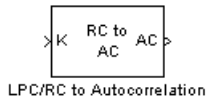
Description   The LPC/RC to Autocorrelation block either converts linear prediction coefficients (LPCs) to autocorrelation coefficients (ACs) or reflection coefficients (RCs) to autocorrelation coefficients (ACs). Set the **Type of conversion** parameter to LPC to autocorrelation or RC to autocorrelation to select the domain from which you want to convert your coefficients. The A port corresponds to LPC coefficients, and the K port corresponds to the RC coefficients.

Use the **Specify P** parameter to set the value of the prediction error power. You can set this parameter to 1 by selecting Assume P=1. When you select Via input port, a P port appears on the block. You can use this port to input the value of the actual, non-unity prediction error power.

The **If first input value is not 1** parameter specifies the behavior of the block when the first coefficient of the LPC coefficient vector is not 1. The following options are available:

- Replace it with 1 — The block changes the first value of the coefficient vector to 1. The rest of the coefficient values are unchanged.

- Normalize — The block divides the entire vector of coefficients by the first coefficient so that the first coefficient of the LPC coefficient vector is 1.

- Normalize and Warn — The block divides the entire vector of coefficients by the first coefficient so that the first coefficient of the LPC coefficient vector is 1. The block displays a warning message telling you that your vector of coefficients has been normalized.

- Error — The block displays an error telling you that the first coefficient of the LPC coefficient vector is not 1.

LPC/RC to Autocorrelation

Dialog Box
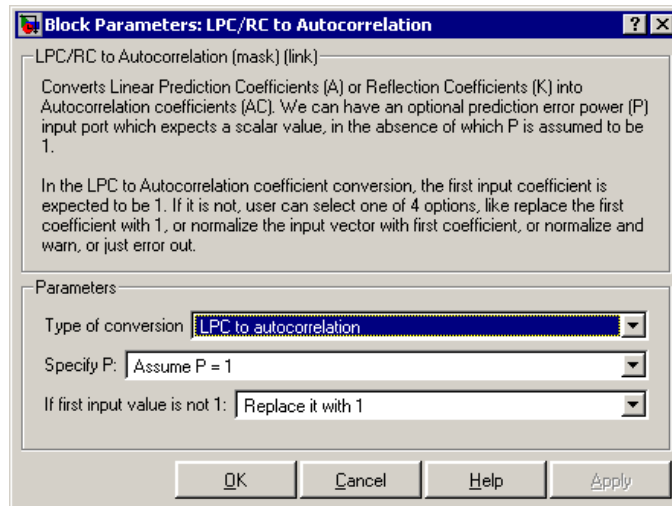
Type of conversion

From the list select LPC to autocorrelation or RC to autocorrelation to specify the domain from which you want to convert your coefficients.

Specify P

From the list select Assume P=1 or Via input port to specify the value of prediction error power.

If first input value is not 1

Select what you would like the block to do when the first coefficient of the LPC coefficient vector is not 1. You can choose Replace it with 1, Normalize, Normalize and Warn, and Error.

References

Orfanidis, S.J. *Optimum Signal Processing*. New York, McGraw-Hill, 1988.

Makhoul, J. *Linear Prediction: A tutorial review*. Proc. IEEE. 63, 63, 56 (1975).

Markel, J.D. and A. H. Gray, Jr., *Linear Prediction of Speech*. New York, Springer-Verlag, 1976.

Supported Data Types

- Double-precision floating point

- Single-precision floating point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Levinson-Durbin	Signal Processing Blockset
LPC to LSF/LSP Conversion	Signal Processing Blockset
LSF/LSP to LPC Conversion	Signal Processing Blockset
LPC to/from RC	Signal Processing Blockset

LU Factorization

Purpose   Factor square matrix into lower and upper triangular components

Library   Math Functions / Matrices and Linear Algebra / Matrix Factorizations

Description

LU Factorization

The LU Factorization block factors a row permutation of the square input matrix A as $A_p = L*U$, where L is the "psychologically lower triangular" matrix, and U is the upper triangular matrix. For more information, see the `lu` function reference page in the MATLAB documentation. The row-pivoted matrix A_p contains the rows of A permuted as indicated by the permutation index vector P.

```
Ap = A(P,:)      % Equivalent
MATLAB code
```

The output of the LU Factorization block at port LU is a composite matrix with lower subtriangle elements from L and upper triangle elements from U. It is always sample based. The output is not in the same form as the output of the MATLAB `lu` function. In order to convert the output of the LU Factorization block to the MATLAB form, use the following equations:

```
L = tril(LU,-1)+diag(ones(size(LU,1),1));
U = triu(LU);
```

Here, `LU` is the output of the LU Factorization block. Due to roundoff error, these equations do not produce a result that is exactly the same as the MATLAB result.

Examples

The row-pivoted matrix A_p and permutation index vector P computed by the block are shown below for 3-by-3 input matrix A.

$$A = \begin{bmatrix} -1 & 8 & -5 \\ 9 & -1 & 2 \\ 2 & -5 & 7 \end{bmatrix} \qquad P = (2\ 1\ 3) \qquad A_p = \begin{bmatrix} 9 & -1 & 2 \\ -1 & 8 & -5 \\ 2 & -5 & 7 \end{bmatrix}$$

The LU output is a composite matrix whose lower subtriangle forms L and whose upper triangle forms U.

$$\begin{bmatrix} -1 & 8 & -5 \\ 9 & -1 & 2 \\ 2 & -5 & 7 \end{bmatrix} \Rightarrow \qquad \begin{bmatrix} 9.00 & -1.00 & 2.00 \\ -0.11 & 7.89 & -4.78 \\ 0.22 & -0.61 & 3.66 \end{bmatrix}$$

$$(2 \quad 1 \quad 3)$$

$$L = \begin{bmatrix} 1 & 0 & 0 \\ -0.11 & 1 & 0 \\ 0.22 & -0.61 & 1 \end{bmatrix} \qquad U = \begin{bmatrix} 9.00 & -1.00 & 2.00 \\ 0 & 7.89 & -4.78 \\ 0 & 0 & 3.66 \end{bmatrix}$$

Dialog Box

Block Parameters: LU Factorization

LU Factorization (mask) (link)

LU factorization with row pivoting. Only for use with a square input matrix A. Stores L (a unit-lower triangular matrix) in the lower triangle and U in the upper triangle of the output matrix. Permutation vector P is output separately. Optionally, the block signals the singularity status of the input matrix

Parameters

☐ Show singularity status

OK   Cancel   Help   Apply

Show singularity status

When selected, the block indicates the singularity of the input at a third output port labeled S, which outputs Boolean data type values of 1 or 0. An output of 1 indicates that the current input is singular, and an output of 0 indicates the current input is nonsingular.

References   Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

LU Factorization

Supported Data Types

Port	Supported Data Types
A	• Double-precision floating point • Single-precision floating point
LU	• Double-precision floating point • Single-precision floating point
P	• Double-precision floating point • Single-precision floating point
S	• Boolean

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Autocorrelation LPC	Signal Processing Blockset
Cholesky Factorization	Signal Processing Blockset
LDL Factorization	Signal Processing Blockset
LU Inverse	Signal Processing Blockset
LU Solver	Signal Processing Blockset
Permute Matrix	Signal Processing Blockset
QR Factorization	Signal Processing Blockset
lu	MATLAB

See "Factoring Matrices" on page 6-9 for related information.

Purpose   Compute inverse of square matrix using LU factorization

Library   Math Functions / Matrices and Linear Algebra / Matrix Inverses

Description   The LU Inverse block computes the inverse of the square input matrix A by factoring and inverting row-pivoted variant A_p.

```
General
Inverse

(LU)
```
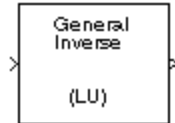
$$A_p^{-1} = (LU)^{-1}$$

L is a lower-triangular square matrix with unity diagonal elements, and U is an upper-triangular square matrix. The block's output is A^{-1}, and is always sample based.
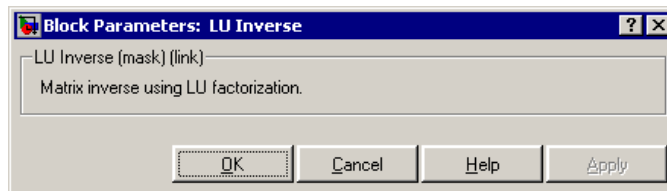
Dialog Box

Block Parameters: LU Inverse

LU Inverse (mask) (link)

Matrix inverse using LU factorization.

OK	Cancel	Help	Apply

References   Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

Supported Data Types

• Double-precision floating point

• Single-precision floating point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Cholesky Inverse	Signal Processing Blockset
LDL Inverse	Signal Processing Blockset

LU Inverse

LU Factorization	Signal Processing Blockset
LU Solver	Signal Processing Blockset
inv	MATLAB

See "Inverting Matrices" on page 6-10 for related information.

Purpose   Solve *AX=B* for *X* when *A* is square matrix

Library   Math Functions / Matrices and Linear Algebra / Linear System Solvers
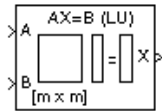
Description The LU Solver block solves the linear system *AX=B* by applying LU factorization to the *M*-by-*M* matrix at the A port. The input to the B port is the right side *M*-by-*N* matrix, *B*. The output is the unique solution of the equations, *M*-by-*N* matrix *X*, and is always sample based.

A length-*M* 1-D vector input for right side *B* is treated as an *M*-by-1 matrix.

Algorithm   The LU algorithm factors a row-permuted variant (A_p) of the square input matrix *A* as

$$A_p = LU$$

where *L* is a lower-triangular square matrix with unity diagonal elements, and *U* is an upper-triangular square matrix.

The matrix factors are substituted for A_p in

$$A_p X = B_p$$

where B_p is the row-permuted variant of *B*, and the resulting equation

$$LUX = B_p$$

is solved for *X* by making the substitution *Y = UX*, and solving two triangular systems.

$$LY = B_p$$

$$UX = Y$$

LU Solver

Dialog Box

Block Parameters: LU Solver

LU Solver (mask)

Solve AX=B using LU decomposition. A must be square. B must have the same number of rows as A.

[OK]   Cancel   Help   Apply

Supported Data Types

- Double-precision floating point
- Single-precision floating point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Autocorrelation LPC	Signal Processing Blockset
Cholesky Solver	Signal Processing Blockset
LDL Solver	Signal Processing Blockset
Levinson-Durbin	Signal Processing Blockset
LU Factorization	Signal Processing Blockset
LU Inverse	Signal Processing Blockset
QR Solver	Signal Processing Blockset

See "Solving Linear Systems" on page 6-7 for related information.

Purpose

Compute nonparametric estimate of the spectrum using the periodogram method

Library

• Estimation / Power Spectrum Estimation

• Transforms

Description
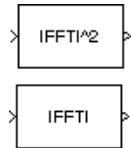
The Magnitude FFT block computes a nonparametric estimate of the spectrum using the periodogram method. When the **Output** parameter is set to Magnitude squared, the block output for an input u is equivalent to

```
y = abs(fft(u,nfft)).^2      % Equivalent
MATLAB code
```

When the **Output** parameter is set to Magnitude, the block output for an input u is equivalent to

```
y = abs(fft(u,nfft))        % Equivalent
MATLAB code
```

Both an M-by-N frame-based matrix input and an M-by-N sample-based matrix input are treated as M sequential time samples from N independent channels. The block computes a separate estimate for each of the N independent channels and generates an N_{fft}-by-N matrix output. When you select **Inherit FFT length from input dimensions**, N_{fft} is specified by the frame size of the input, which must be a power of 2. When you do *not* select **Inherit FFT length from input dimensions**, N_{fft} is specified as a power of 2 by the **FFT length** parameter, and the block zero pads or truncates the input to N_{fft} before computing the FFT.

Each column of the output matrix contains the estimate of the corresponding input column's power spectral density at N_{fft} equally spaced frequency points in the range $[0,F_{\mathrm{s}})$, where F_{s} is the signal's sample frequency. The output is always sample based.

The block does not accept sample-based 1-by-N row vector inputs.

Magnitude FFT

The Magnitude FFT block supports real and complex floating-point and fixed-point inputs.

Fixed-Point Data Types

The following diagram shows the data types used within the Magnitude FFT subsystem block for fixed-point signals.
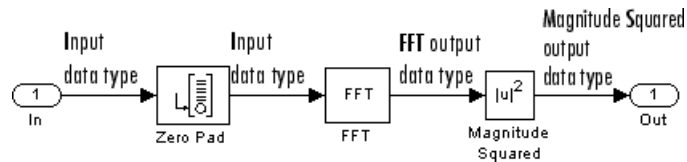
The settings for the fixed-point parameters of the FFT block in the diagram above are as follows:

- Sine table — `Same word length as input`
- Round integer calculations toward: `Floor`
- Saturate on integer overflow — unchecked
- Product output — `Inherit via internal rule`
- Accumulator — `Inherit via internal rule`
- Output — `Inherit via internal rule`

The settings for the fixed-point parameters of the Magnitude Squared block in the diagram above are as follows:

- Round integer calculations toward: `Floor`
- Saturate on integer overflow — checked
- Output — `Inherit via internal rule`

The Magnitude Squared block is an implementation of the Simulink Math Function block. Refer to the FFT, Zero Pad, and Math Function reference pages for more information.

Examples   The dspsacomp demo compares the periodogram method with several
other spectral estimation methods.

**Dialog
Box**
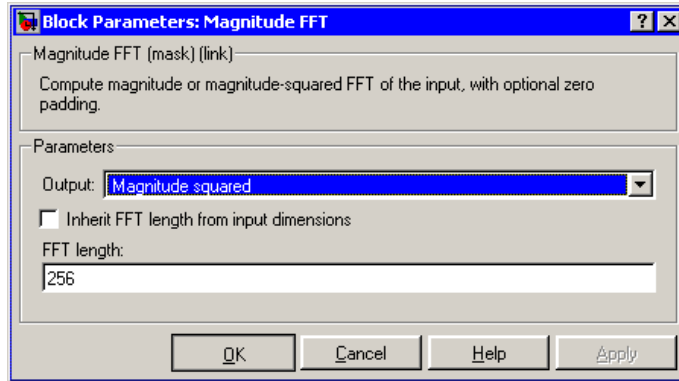
Output

> Determines whether the block computes the magnitude FFT
> (Magnitude) or magnitude-squared FFT (Magnitude squared) of
> the input. Nontunable.

Inherit FFT length from input dimensions

> When selected, uses the input frame size as the number of data
> points, N_{fft}, on which to perform the FFT.

FFT length

> The number of data points on which to perform the FFT, N_{fft}.
> When N_{fft} exceeds the input frame size, the frame is zero-padded
> as needed. This parameter is enabled when you do not select
> **Inherit FFT length from input dimensions**.

References   Oppenheim, A. V. and R. W. Schafer. *Discrete-Time Signal Processing*.
Englewood Cliffs, NJ: Prentice Hall, 1989.

Proakis, J. and D. Manolakis. *Digital Signal Processing*. 3rd ed.
Englewood Cliffs, NJ: Prentice-Hall, 1996.

Magnitude FFT

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point
- 8-, 16-, and 32-bit signed integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Burg Method	Signal Processing Blockset
Short-Time FFT	Signal Processing Blockset
Spectrum Scope	Signal Processing Blockset
Yule-Walker Method	Signal Processing Blockset
pwelch	Signal Processing Toolbox

See "Power Spectrum Estimation" on page 6-6 for related information.

Purpose   Compute the 1-norm of a matrix

Library   Math Functions / Matrices and Linear Algebra / Matrix Operations

Description   The Matrix 1-Norm block computes the 1-norm, or maximum column-sum, of an *M*-by-*N* input matrix, *A*.

$$y = \|A\|_1 = \max_{1 \le j \le N} \sum_{i=1}^{M} |a_{ij}|$$

This is equivalent to

```
y = max(sum(abs(A)))       % Equivalent
MATLAB code
```
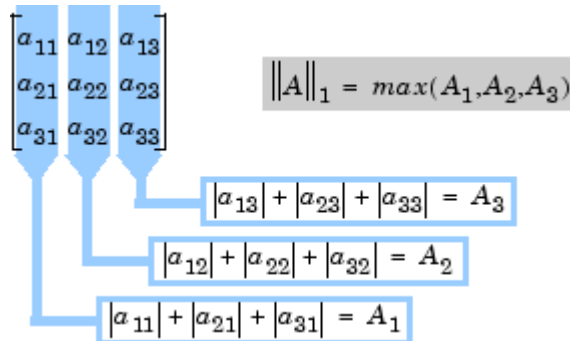

A length-*M* 1-D vector input is treated as an *M*-by-1 matrix. The output, *y*, is always a scalar.

The Matrix 1-Norm block supports real and complex floating-point inputs, and real fixed-point inputs.

Fixed-Point Data Types

The following diagram shows the data types used within the Matrix 1-Norm block for fixed-point signals.
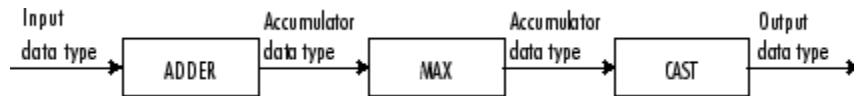
The block calculations are all done in the accumulator data type until the max is performed. The result is then cast to the output data type. You can set the accumulator and output data types in the block dialog as discussed in "Dialog Box" on page 10-666 below.

Dialog Box

There are no parameters on the **Main** pane of this dialog.

The **Fixed-point** pane of the Matrix 1-Norm block dialog appears as follows:
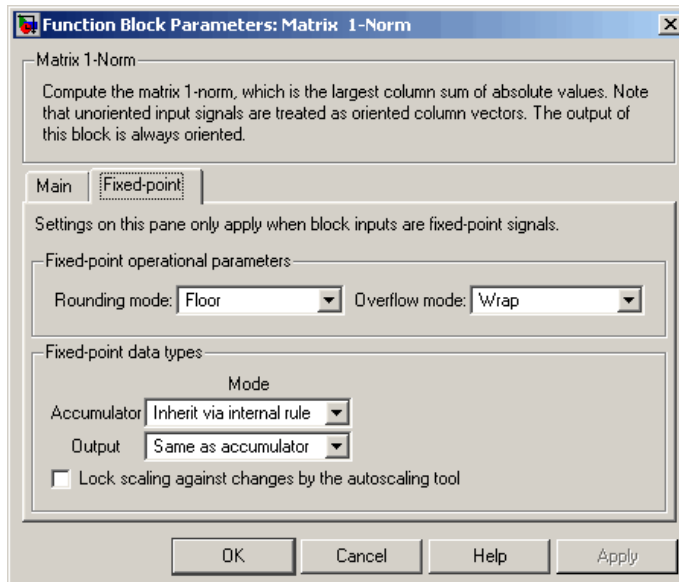
Rounding mode
Select the rounding mode for fixed-point operations.

Overflow mode
Select the overflow mode for fixed-point operations.

Accumulator

Choose how you will specify the word length and fraction length of the accumulator:

- When you select `Inherit via internal rule`, the accumulator word length and fraction length are automatically set according to the following equations:

$$ideal\ accumulator\ word\ length =$$
$$input\ word\ length + \text{floor}(\log 2(number\ of\ columns)) + 1$$

$$ideal\ accumulator\ fraction\ length\ =\ input\ fraction\ length$$

> **Note** The actual accumulator word length may be equal to or greater than the calculated ideal product output word length, depending on the settings on the **Hardware Implementation** pane of the Configuration Parameters dialog box.

- When you select `Same as input`, these characteristics will match those of the input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the accumulator, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the accumulator. This block requires power-of-two slope and a bias of zero.

Output

Choose how you will specify the word length and fraction length of the output of the block:

- When you select `Same as input`, these characteristics will match those of the input to the block.

- When you select `Same as accumulator`, these characteristics will match those of the accumulator.

Matrix 1-Norm

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the output, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the output. This block requires power-of-two slope and a bias of zero.

Lock scaling against changes by the autoscaling tool
Select this parameter to prevent any fixed-point scaling you specify in this block mask from being overriden by the autoscaling tool in the Fixed-Point Settings interface. For more information about the autoscaling tool, refer to "Fixed-Point Settings Interface" on page 8-28.

References Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

Supported Data Types
- Double-precision floating point
- Single-precision floating point
- Fixed point (signed only)
- 8-, 16-, and 32-bit signed integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Normalization	Signal Processing Blockset
Reciprocal Condition	Signal Processing Blockset
norm	MATLAB

Purpose   Compute matrix exponential

Library   Math Functions / Matrices and Linear Algebra / Matrix Operations

Description   The Matrix Exponential block computes the matrix exponential using a scaling and squaring algorithm with a Pade approximation. The input matrix must be square.
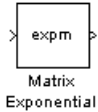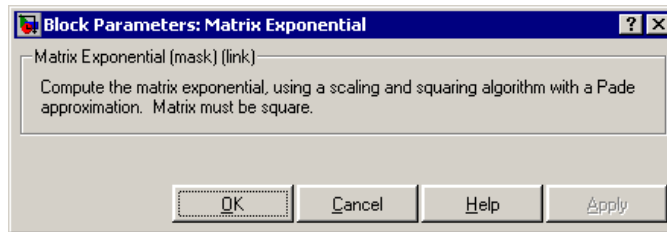
> expm >

Matrix
Exponential

Dialog Box

Block Parameters: Matrix Exponential   ? ✕

Matrix Exponential (mask) (link)

Compute the matrix exponential, using a scaling and squaring algorithm with a Pade approximation. Matrix must be square.

    OK   Cancel   Help   Apply

Supported Data Types

- Double-precision floating point
- Single-precision floating point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

expm	MATLAB
Dot Product	Simulink
Matrix Product	Signal Processing Blockset
Matrix Scaling	Signal Processing Blockset
Product	Simulink

Matrix Multiply

Purpose   Multiply or divide inputs

Library   Math Functions / Matrices and Linear Algebra / Matrix Operations

Description   The Matrix Multiply block is an implementation of the Simulink Product block. See Product for more information.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point
- Boolean
- 8–, 16–, and 32–bit signed integers
- 8–, 16–, and 32–bit unsigned integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Dot Product	Simulink
Matrix Product	Signal Processing Blockset
Matrix Scaling	Signal Processing Blockset
Product	Simulink

Purpose   Multiply matrix elements along rows or columns

Library   Math Functions / Matrices and Linear Algebra / Matrix Operations

Description   The Matrix Product block multiplies the elements of an *M*-by-*N* input
matrix *u* along either the rows or columns. When the **Multiply along**
parameter is set to Rows, the block multiplies across the elements of
each row and outputs the resulting *M*-by-1 matrix. A length-*N* 1-D
vector input is treated as a 1-by-*N* matrix.

$$
\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \\ u_{31} & u_{32} & u_{33} \end{bmatrix} \implies \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} (u_{11}u_{12}u_{13}) \\ (u_{21}u_{22}u_{23}) \\ (u_{31}u_{32}u_{33}) \end{bmatrix}
$$

This is equivalent to

```
y = prod(u,2)     % Equivalent MATLAB code
```

When the **Multiply along** parameter is set to Columns, the block
multiplies down the elements of each column and outputs the resulting
1-by-*N* matrix. A length-*M* 1-D vector input is treated as a *M*-by-1
matrix.

$$
\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \\ u_{31} & u_{32} & u_{33} \end{bmatrix}
$$

$$
\begin{bmatrix} y_1 & y_2 & y_3 \end{bmatrix} = \begin{bmatrix} (u_{11}u_{21}u_{31}) & (u_{12}u_{22}u_{32}) & (u_{13}u_{23}u_{33}) \end{bmatrix}
$$

This is equivalent to

```
y = prod(u)     % Equivalent MATLAB code
```

Matrix Product

The output of the Matrix Product block has the same frame status as the input. This block accepts real and complex floating-point and fixed-point inputs.

Fixed-Point Data Types

The following diagram shows the data types used within the Matrix Product block for fixed-point signals.
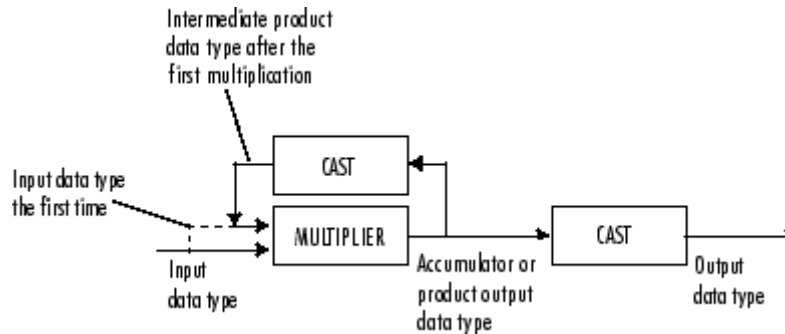
The output of the multiplier is in the product output data type when at least one of the inputs to the multiplier is real. When both of the inputs to the multiplier are complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed, refer to "Multiplication Data Types" on page 8-16. You can set the accumulator, product output, intermediate product, and output data types in the block dialog as discussed in "Dialog Box" on page 10-673 below.

Dialog Box

The **Main** pane of the Matrix Product block dialog appears as follows:
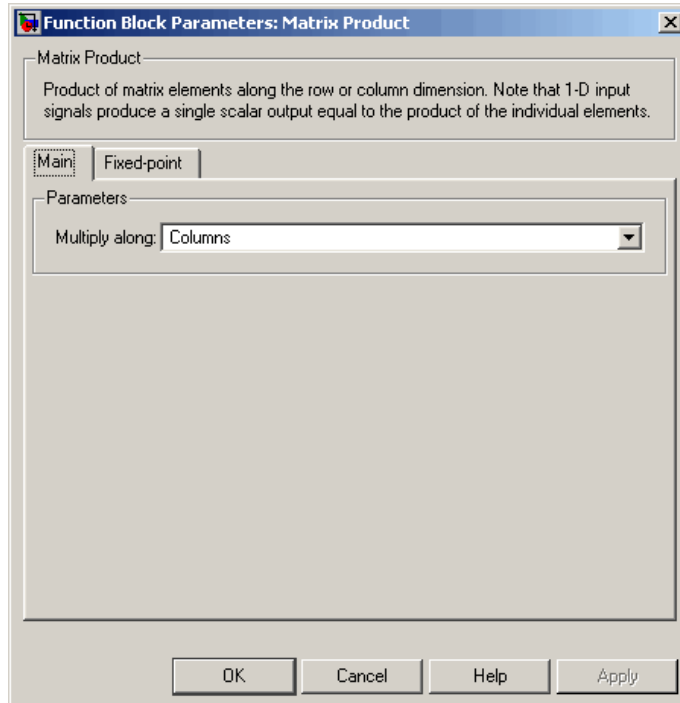
Multiply along

Indicate whether to multiply together the elements of each row or of each column of the input.

The **Fixed-point** pane of the Matrix Product block dialog appears as follows:

Matrix Product
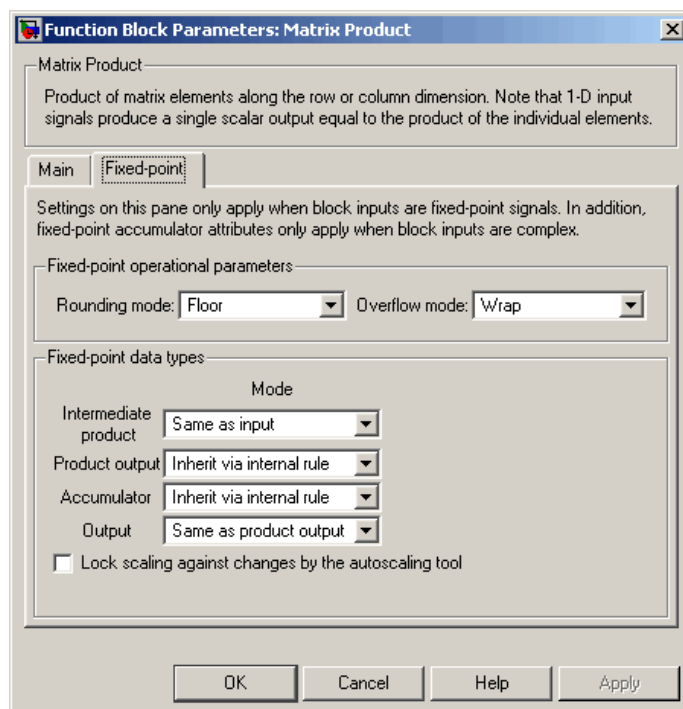
Rounding mode

Select the rounding mode for fixed-point operations.

Overflow mode

Select the overflow mode for fixed-point operations.

Intermediate product

As shown in "Fixed-Point Data Types" on page 10-672, the output of the multiplier is cast to the intermediate product data type before the next element of the input is multiplied into it. Use this parameter to specify how you would like to designate the intermediate product word and fraction lengths:

- When you select Same as input, these characteristics will match those of the input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the intermediate product, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the intermediate product. This block requires power-of-two slope and a bias of zero.

Product output

Use this parameter to specify how you would like to designate the product output word and fraction lengths. Refer to "Fixed-Point Data Types" on page 10-672 and "Multiplication Data Types" on page 8-16 for illustrations depicting the use of the product output data type in this block:

- When you select `Inherit via internal rule`, the product output word length and fraction length are automatically set according to the following equations:

$$ideal\ product\ output\ word\ length\ = \\ input\ word\ length + intermediate\ product\ word\ length$$

$$ideal\ product\ output\ fraction\ length\ = \\ input\ fraction\ length + intermediate\ product\ fraction\ length$$

Note The actual product output word length may be equal to or greater than the calculated ideal product output word length, depending on the settings on the **Hardware Implementation** pane of the Configuration Parameters dialog box.

- When you select `Same as input`, these characteristics will match those of the input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the product output, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the product output. This block requires power-of-two slope and a bias of zero.

Accumulator

Use this parameter to specify how you would like to designate the accumulator word and fraction lengths. Refer to "Fixed-Point Data Types" on page 10-672 and "Multiplication Data Types" on page 8-16 for illustrations depicting the use of the accumulator data type in this block. Note that the accumulator data type is only used when both inputs to the multiplier are complex:

- When you select `Inherit via internal rule`, the accumulator word length and fraction length are automatically set according to the following equations:

$$ideal\ accumulator\ word\ length = \\ ideal\ product\ output\ word\ length + 1$$

$$ideal\ accumulator\ fraction\ length = \\ ideal\ product\ output\ fraction\ length$$

Note The actual accumulator word length may be equal to or greater than the calculated ideal product output word length, depending on the settings on the **Hardware Implementation** pane of the Configuration Parameters dialog box.

- When you select `Same as product output`, these characteristics will match those of the product output.

- When you select `Same as input`, these characteristics will match those of the input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the accumulator, in bits.

- When you select Slope and bias scaling, you are able to enter the word length, in bits, and the slope of the accumulator. This block requires power-of-two slope and a bias of zero.

Output

Choose how you will specify the word length and fraction length of the output of the block:

- When you select Same as product output, these characteristics will match those of the product output.

- When you select Same as input, these characteristics will match those of the input to the block.

- When you select Binary point scaling, you are able to enter the word length and the fraction length of the output, in bits.

- When you select Slope and bias scaling, you are able to enter the word length, in bits, and the slope of the output. This block requires power-of-two slope and a bias of zero.

Lock scaling against changes by the autoscaling tool

Select this parameter to prevent any fixed-point scaling you specify in this block mask from being overriden by the autoscaling tool in the Fixed-Point Settings interface. For more information about the autoscaling tool, refer to "Fixed-Point Settings Interface" on page 8-28.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point (signed and unsigned)
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

Matrix Product

See Also

Matrix Multiply	Signal Processing Blockset
Matrix Square	Signal Processing Blockset
Matrix Sum	Signal Processing Blockset
prod	MATLAB

Purpose   Scale matrix rows or columns by specified vector

Library   Math Functions / Matrices and Linear Algebra / Matrix Operations
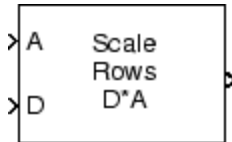
Description   The Matrix Scaling block scales the rows or columns of the M-by-N input matrix A by the values in input vector D. When the **Mode** parameter is set to Scale Rows (D*A), the input D can be a 1-D or 2-D vector of length M, and the block multiplies each element of D across the corresponding *row* of matrix A.

$$\begin{bmatrix} d_1 \\ d_2 \\ d_3 \end{bmatrix} \times \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \Rightarrow \begin{bmatrix} d_1a_{11} & d_1a_{12} & d_1a_{13} \\ d_2a_{21} & d_2a_{22} & d_2a_{23} \\ d_3a_{31} & d_3a_{32} & d_3a_{33} \end{bmatrix}$$

This is equivalent to premultiplying A by a diagonal matrix with diagonal D.

```
y = diag(D)*A     % Equivalent MATLAB code
```

When the **Mode** parameter is set to Scale Columns (A*D), the input D can be a 1-D or 2-D vector of length N, and the block multiplies each element of D across the corresponding *column* of matrix A.

$$\begin{bmatrix} d_1 & d_2 & d_3 \end{bmatrix} \\ \times \quad \times \quad \times \\ \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \Rightarrow \begin{bmatrix} d_1a_{11} & d_2a_{12} & d_3a_{13} \\ d_1a_{21} & d_2a_{22} & d_3a_{23} \\ d_1a_{31} & d_2a_{32} & d_3a_{33} \end{bmatrix}$$

This is equivalent to postmultiplying A by a diagonal matrix with diagonal D.

```
y = A*diag(D)     % Equivalent MATLAB code
```

Matrix Scaling

The output of the Matrix Scaling block is the same size as the input matrix, A. When both inputs are sample based, the output is sample based; otherwise, the output is frame based. This block accepts real and complex floating-point and fixed-point inputs.

Fixed-Point Data Types

The following diagram shows the data types used within the Matrix Scaling block for fixed-point signals.
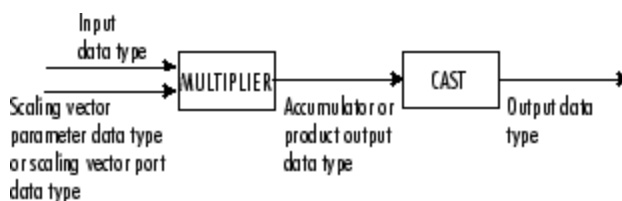
When the scaling vector D is designated in the block mask, its elements have the data type and scaling that you specify in the **Scaling vector** parameters on the **Fixed-point** tab. When the scaling vector comes in through the block port, its elements inherit their data type and scaling from the driving block.

The output of the multiplier is in the product output data type when at least one of the inputs to the multiplier is real. When both of the inputs to the multiplier are complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed, refer to "Multiplication Data Types" on page 8-16.

You can set the scaling vector, accumulator, product output, and output data types in the block dialog as discussed below.

Dialog Box

The **Main** pane of the Matrix Scaling block dialog appears as follows:
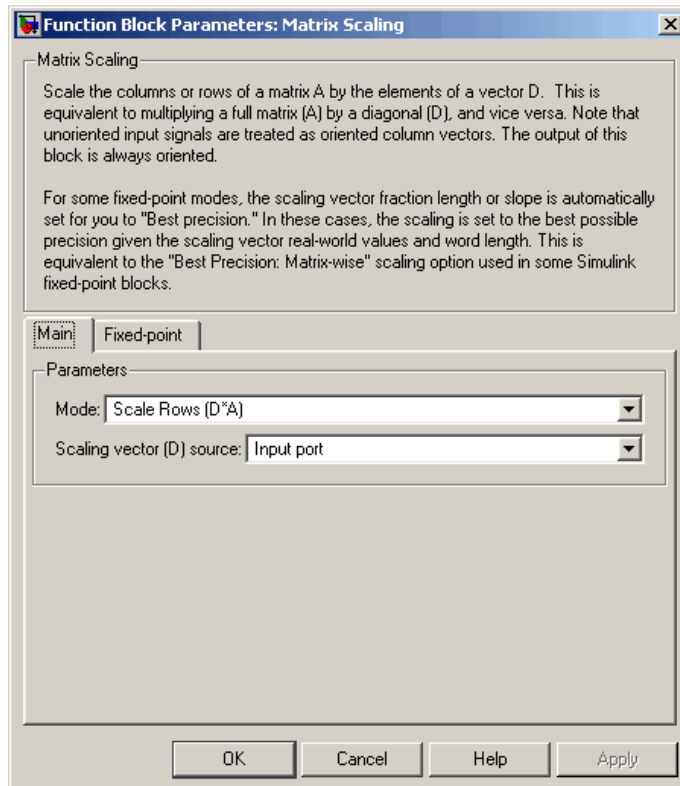
Mode

Specify the mode of operation, row scaling or column scaling. Nontunable.

Scaling vector (D) source

Specify the source of the scaling vector, D. The scaling vector can come from an Input port or from a Dialog parameter.

Scaling vector (D)

Specify the scaling vector, D. This parameter is visible only when you select Input port for the **Scaling vector (D) source** parameter.

The **Fixed-point** pane of the Matrix Scaling block dialog appears as follows:
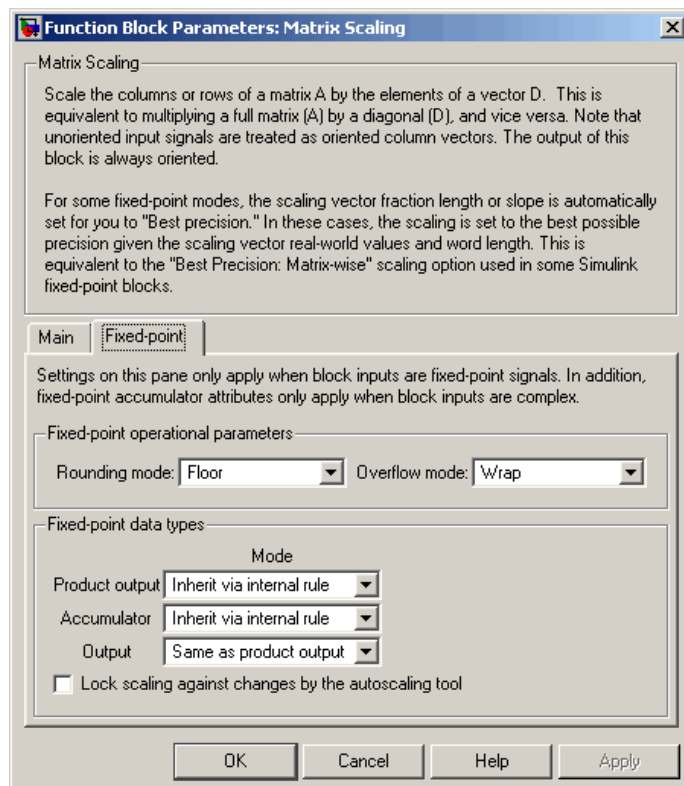
Rounding mode

Select the rounding mode for fixed-point operations.

Overflow mode

Select the overflow mode for fixed-point operations.

Scaling vector

Use this parameter to specify how you would like to designate the word and fraction lengths of the elements of the scaling vector, *D*:

- When you select `Same word length as input`, the word length of the scaling vector values will match that of the input to the block.

- When you select `Specify word length`, you are able to enter the word length of the scaling vector values, in bits. In this mode, the fraction length of the scaling vector values is automatically set to the binary-point only scaling that provides you with the best precision possible given the value and word length of the values.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the scaling vector elements, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the scaling vector element. This block requires power-of-two slope and a bias of zero.

Note The **Scaling vector** parameters on the **Fixed-point** pane are only applicable when you specify the scaling vector through the **Scaling vector (D)** parameter on the block mask. When the scaling vector comes in through the block port, the data type and scaling of its elements are inherited from the driving block.

Product output

Use this parameter to specify how you would like to designate the product output word and fraction lengths. Refer to "Fixed-Point Data Types" on page 10-680 and "Multiplication Data Types" on

Matrix Scaling

page 8-16 for illustrations depicting the use of the product output data type in this block:

- When you select `Inherit via internal rule`, the product output word length and fraction length are automatically set according to the following equations:

$$ideal\ product\ output\ word\ length\ =$$
$$word\ length\ of\ first\ input + word\ length\ of\ scaling\ vector\ coefficients$$

$$ideal\ product\ output\ fraction\ length\ =$$
$$fraction\ length\ of\ first\ input +$$
$$fraction\ length\ of\ scaling\ vector\ coefficients$$

Note The actual product output word length may be equal to or greater than the calculated ideal product output word length, depending on the settings on the **Hardware Implementation** pane of the Configuration Parameters dialog box.

- When you select `Same as first input`, these characteristics will match those of the first input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the product output, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the product output. This block requires power-of-two slope and a bias of zero.

Accumulator
Use this parameter to specify how you would like to designate the accumulator word and fraction lengths. Refer to "Fixed-Point Data Types" on page 10-680 and "Multiplication Data Types" on page 8-16 for illustrations depicting the use of the accumulator data type in this block. Note that the accumulator data type is only used when both inputs to the multiplier are complex:

- When you select `Inherit via internal rule`, the accumulator word length and fraction length are automatically set according to the following equations:

$$ideal\ accumulator\ word\ length = \\ ideal\ product\ output\ word\ length + 1$$

$$ideal\ accumulator\ fraction\ length = \\ ideal\ product\ output\ fraction\ length$$

Note The actual accumulator word length may be equal to or greater than the calculated ideal product output word length, depending on the settings on the **Hardware Implementation** pane of the Configuration Parameters dialog box.

- When you select `Same as product output`, these characteristics will match those of the product output.

- When you select `Same as first input`, these characteristics will match those of the first input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the accumulator, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the accumulator. This block requires power-of-two slope and a bias of zero.

Output

Choose how you will specify the output word length and fraction length:

- When you select `Same as accumulator`, these characteristics will match those of the accumulator.

- When you select `Same as product output`, these characteristics will match those of the product output.

- When you select `Same as first input`, these characteristics will match those of the first input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the output, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the output. This block requires power-of-two slope and a bias of zero.

Lock scaling against changes by the autoscaling tool
Select this parameter to prevent any fixed-point scaling you specify in this block mask from being overriden by the autoscaling tool in the Fixed-Point Settings interface. For more information about the autoscaling tool, refer to "Fixed-Point Settings Interface" on page 8-28.

Supported Data Types

- Double-precision floating point

- Single-precision floating point

- Fixed point (signed only)

- 8-, 16-, and 32-bit signed integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Matrix Multiply	Signal Processing Blockset
Matrix Product	Signal Processing Blockset
Matrix Sum	Signal Processing Blockset

Purpose   Compute square of input matrix

Library   Math Functions / Matrices and Linear Algebra / Matrix Operations
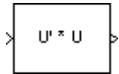
Description   The Matrix Square block computes the square of an *M*-by-*N* input
                  matrix, *u*, by premultiplying with the Hermitian transpose.

```
U' * U
```

```
y = u' * u      % Equivalent MATLAB code
```

A length-*M* 1-D vector input is treated as an *M*-by-1 matrix. For both
sample-based and frame-based inputs, output *y* is sample based with
dimension *N*-by-*N*.

Applications

The Matrix Square block is useful in a variety of applications:

- *General matrix squares* — The Matrix Square block computes the
  output matrix, *y*, without explicitly forming *u'*. It is therefore more
  efficient than other methods for computing the matrix square.

- *Sum of squares* — When the input is a column vector (*N*=1), the
  block's operation is equivalent to a multiply-accumulate (MAC)
  process, or inner product. The output is the sum of the squares of the
  input, and is always a real scalar.

- *Correlation matrix* — When the input is a row vector (*M*=1), the
  output, *y*, is the symmetric autocorrelation matrix, or outer product.

Dialog Box

Block Parameters: Matrix Square

Matrix Square (mask) (link)

Compute the matrix square, U' * U. Hermitian transpose is performed for complex
inputs. [Mx1] and [1xN] input matrices may be used to compute efficient outer and
inner products, respectively. Vector input signals are treated as [Mx1] matrices. The
output is always a matrix.

OK   Cancel   Help   Apply
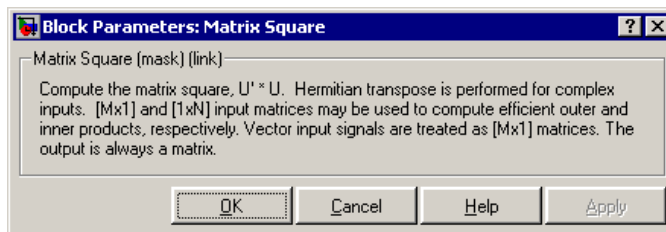
Matrix Square

Supported Data Types

- Double-precision floating point
- Single-precision floating point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Matrix Multiply	Signal Processing Blockset
Matrix Product	Signal Processing Blockset
Matrix Sum	Signal Processing Blockset
Transpose	Signal Processing Blockset

Purpose   Sum matrix elements along rows or columns

Library   Math Functions / Matrices and Linear Algebra / Matrix Operations

Description

Row
Sum
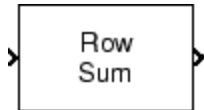
The Matrix Sum block sums the elements of an M-by-N input matrix u along either the rows or columns. When the **Sum along** parameter is set to Rows, the block sums across the elements of each row and outputs the resulting M-by-1 matrix. A length-N 1-D vector input is treated as a 1-by-N matrix.

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \\ u_{31} & u_{32} & u_{33} \end{bmatrix} \quad \Longrightarrow \quad \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} u_{11} + u_{12} + u_{13} \\ u_{21} + u_{22} + u_{23} \\ u_{31} + u_{32} + u_{33} \end{bmatrix}$$

This is equivalent to

```
y = sum(u,2)     % Equivalent MATLAB code
```

When the **Sum along** parameter is set to Columns, the block sums down the elements of each column and outputs the resulting 1-by-N matrix. A length-M 1-D vector input is treated as a M-by-1 matrix.

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \\ u_{31} & u_{32} & u_{33} \end{bmatrix}$$

$$\begin{bmatrix} y_1 & y_2 & y_3 \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{3} u_{i1} & \sum_{i=1}^{3} u_{i2} & \sum_{i=1}^{3} u_{i3} \end{bmatrix}$$

This is equivalent to

```
y = sum(u)       % Equivalent MATLAB code
```

Matrix Sum

The output of the Matrix Sum block has the same frame status as the input. This block accepts real and complex floating-point and fixed-point inputs.

Fixed-Point Data Types

The following diagram shows the data types used within the Matrix Sum block for fixed-point signals.
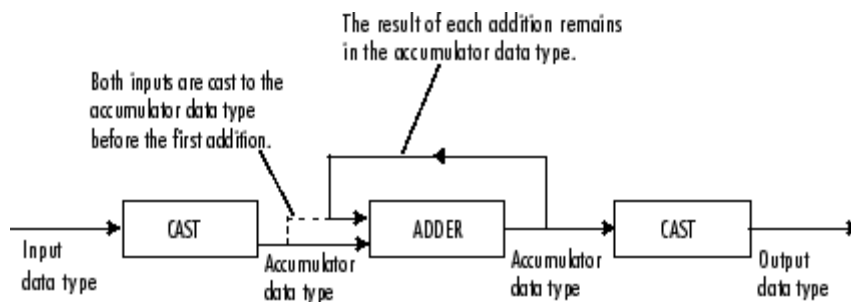
You can set the accumulator and output data types in the block dialog as discussed in "Dialog Box" on page 10-691 below.

Dialog Box

The **Main** pane of the Matrix Sum block dialog appears as follows:
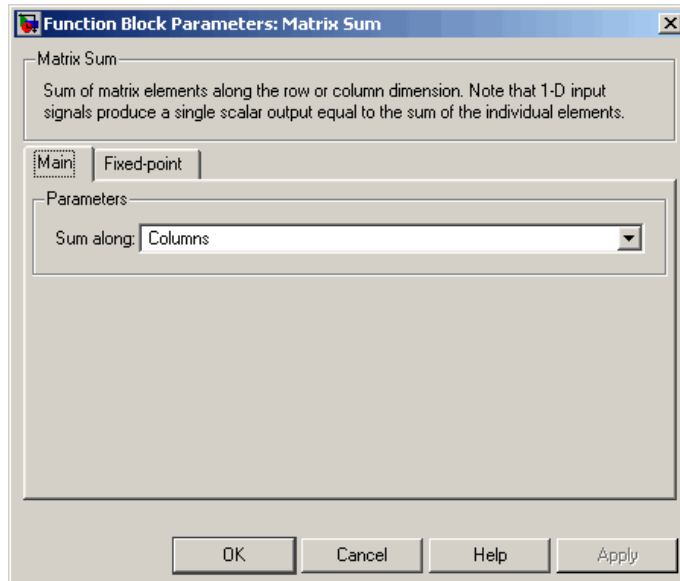
Sum along

 Indicate whether to sum the elements of each row or of each column of the input.

The **Fixed-point** pane of the Matrix Sum block dialog appears as follows:

Matrix Sum
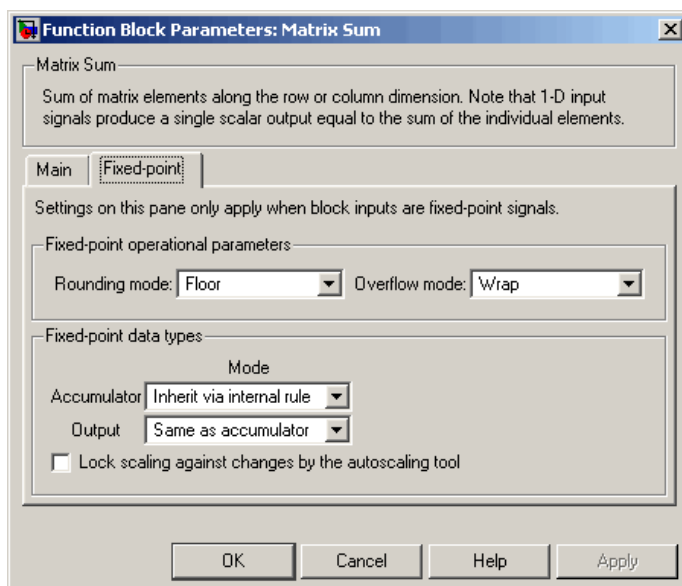
Rounding mode
> Select the rounding mode for fixed-point operations.

Overflow mode
> Select the overflow mode for fixed-point operations.
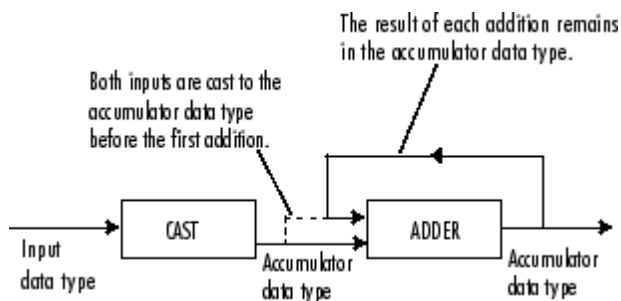
Accumulator

As depicted above, the elements of the block input are cast to the accumulator data type before they are added together. The output of the adder remains in the accumulator data type as each element of the input is added to it. Use this parameter to specify how you would like to designate this accumulator word and fraction lengths:

- When you select `Inherit via internal rule`, the accumulator word length and fraction length are automatically set according to the following equations:

$$ideal\ accumulator\ word\ length =$$
$$input\ word\ length + \text{floor}(\log2(number\ of\ rows\ or\ columns - 1)) + 1$$

$$ideal\ accumulator\ fraction\ length = input\ fraction\ length$$

- When you select `Same as input`, these characteristics will match those of the input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the accumulator, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the accumulator. This block requires power-of-two slope and a bias of zero.

Output

Choose how you will specify the output word length and fraction length:

- When you select `Same as accumulator`, these characteristics will match those of the accumulator.

- When you select `Same as input`, these characteristics will match those of the input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the output, in bits.

Matrix Sum

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the output. This block requires power-of-two slope and a bias of zero.

Lock scaling against changes by the autoscaling tool
Select this parameter to prevent any fixed-point scaling you specify in this block mask from being overriden by the autoscaling tool in the Fixed-Point Settings interface. For more information about the autoscaling tool, refer to "Fixed-Point Settings Interface" on page 8-28.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point (signed and unsigned)
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Matrix Product	Signal Processing Blockset
Matrix Multiply	Signal Processing Blockset
sum	MATLAB

Purpose   Display matrices as color images

Library   Signal Processing Sinks

Description   The Matrix Viewer block displays an *M*-by-*N* matrix input by mapping
the matrix element values to a specified range of colors. The display is
updated as each new input is received. This block treats a length *M* 1-D
vector input as an *M*-by-1 matrix.

Image Properties

Select the **Image Properties** tab to show the image property
parameters, which control the colormap and display.

You specify the mapping of matrix element values to colors in the
Colormap matrix, **Minimum input value**, and **Maximum input
value** parameters. For a colormap with *L* colors, the colormap matrix
has dimension *L*-by-3, with one row for each color and one column for
each element of the RGB triple that defines the color. Examples of RGB
triples are

```
[ 1   0   0 ]   (red)
[ 0   0   1 ]   (blue)
[0.8 0.8 0.8]   (light gray)
```

See the `ColorSpec` property in the MATLAB documentation for
complete information about defining RGB triples.

MATLAB provides a number of functions for generating predefined
colormaps, such as `hot`, `cool`, `bone`, and `autumn`. Each of these
functions accepts the colormap size as an argument, and can be
used in the **Colormap matrix** parameter. For example, when you
specify `gray(128)` for the **Colormap matrix** parameter, the matrix
is displayed in 128 shades of gray. The color in the first row of the
colormap matrix represents the value specified by the **Minimum
input value** parameter, and the color in the last row represents the
value specified by the **Maximum input value** parameter. Values
between the minimum and maximum are quantized and mapped to the
intermediate rows of the colormap matrix.

Matrix Viewer

The documentation for the MATLAB `colormap` function provides complete information about specifying colormap matrices, and includes a complete list of the available colormap functions.

Axis Properties

Select the **Axis Properties** tab to show the axis property parameters, which control labeling and positioning.

The **Axis origin** parameter determines where the first element of the input matrix, $U(1,1)$, is displayed. When you specify `Upper left corner`, the matrix is displayed in *matrix orientation*, with $U(1,1)$ in the upper-left corner.

$$\begin{bmatrix} U_{11} & U_{12} & U_{13} & U_{14} \\ U_{21} & U_{22} & U_{23} & U_{24} \\ U_{31} & U_{32} & U_{33} & U_{34} \\ U_{41} & U_{42} & U_{43} & U_{44} \end{bmatrix}$$

When you specify `Lower left corner`, the matrix is flipped vertically to *image orientation*, with $U(1,1)$ in the lower-left corner.

$$\begin{bmatrix} U_{41} & U_{42} & U_{43} & U_{44} \\ U_{31} & U_{32} & U_{33} & U_{34} \\ U_{21} & U_{22} & U_{23} & U_{24} \\ U_{11} & U_{12} & U_{13} & U_{14} \end{bmatrix}$$

Axis zoom, when selected, causes the image display to completely fill the figure window. Axis titles are not displayed. This option can also be selected from the pop-up menu that is displayed when you right-click in the figure window. When **Axis zoom** is cleared, the axis labels and titles are displayed in a gray border surrounding the image axes.

Figure Window

The image title in the figure title bar is the same as the block title. The axis tick marks reflect the size of the input matrix; the *x*-axis is numbered from 1 to *N* (number of columns), and the *y*-axis is numbered from 1 to *M* (number of rows).

Right-click the image in the figure window to access the following menu items:

- **Refresh** erases all data on the scope display except for the most recent image.

- **Autoscale** recomputes the minimum and maximum input values to fit the range of values observed in a series of 10 consecutive inputs. The numerical limits selected by the autoscale feature are shown in the **Minimum input value** and **Maximum input value** parameters, where you can make further adjustments to them manually.

- **Axis zoom**, when selected, causes the image to completely fill the figure window. Axis titles are not displayed. When **Axis zoom** is cleared, the axis labels and titles are displayed in a gray border surrounding the scope axes. This option can also be set in the Axis Properties pane of the parameter dialog box.

- **Colorbar**, when selected, displays a bar with the specified colormap to the right of the image axes.

- **Save Position** automatically updates the **Figure position** parameter in the **Axis Properties** pane to reflect the figure window's current position and size on the screen. To make the scope window open at a particular location on the screen when the simulation runs, drag the window to the desired location, resize it, and select **Save Position**. The parameter dialog box must be closed when you select **Save Position** for the **Figure position** parameter to be updated.

Examples See the demo dspstfft.mdl for an example of using the Matrix Viewer block to create a moving spectrogram, or time-frequency plot, of a

Matrix Viewer

speech signal by updating just one column of the input matrix at each sample time.

Dialog Box
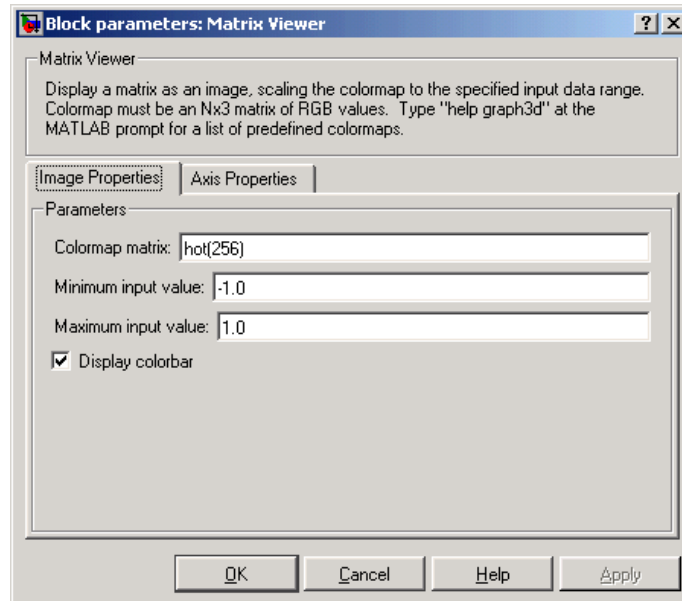
Colormap matrix

A 3-column matrix defining the colormap as a set of RGB triples, or a call to a colormap-generating function such as hot or spring. See the ColorSpec property for complete information about defining RGB triples, and the MATLAB colormap function for a list of colormap-generating functions. Tunable.

Minimum input value

The input value to be mapped to the color defined in the first row of the colormap matrix. Right-click in the figure window and select Autoscale from pop-up menu to set this parameter to the minimum value observed in a series of 10 consecutive matrix inputs. Tunable.

Maximum input value

The input value to be mapped to the color defined in the last row of the colormap matrix. Right-click in the figure window and select Autoscale from the pop-up menu to set this parameter to the maximum value observed in a series of 10 consecutive matrix inputs. Tunable.

Display colorbar

Select to display a bar with the selected colormap to the right of the image axes. Tunable.
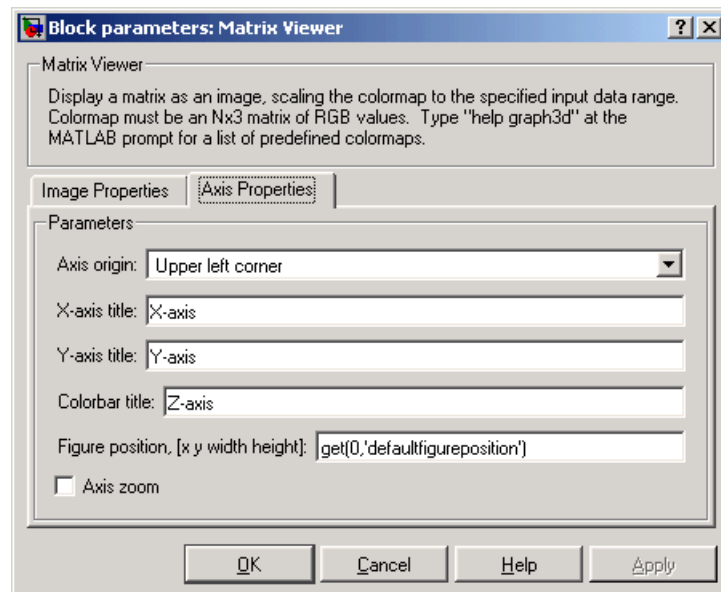
Axis origin

The position within the axes where the first element of the input matrix, $U(1,1)$, is plotted; bottom left or top left. Tunable.

X-axis title

The text to be displayed below the x-axis. Tunable.

Y-axis title

The text to be displayed to the left of the *y*-axis. Tunable.

Colorbar title

The text to be displayed to the right of the color bar, when **Display colorbar** is currently selected. Tunable.

Figure position, [x y width height]

A 4-element vector of the form [x y width height] specifying the position of the figure window, where (0,0) is the lower-left corner of the display. Tunable.

Axis zoom

Resizes the image to fill the figure window. Tunable.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Custom data types
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Spectrum Scope	Signal Processing Blockset
Vector Scope	Signal Processing Blockset

colormap	MATLAB
ColorSpec	MATLAB
image	MATLAB

Maximum

Purpose　　　Find maximum values in an input or sequence of inputs

Library　　　Statistics

Description
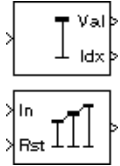
The Maximum block identifies the value and/or position of the largest element in each column of the input, or tracks the maximum values in a sequence of inputs over a period of time. The **Mode** parameter specifies the block's mode of operation and can be set to `Value`, `Index`, `Value and Index`, or `Running`.

The Maximum block supports real and complex floating-point and fixed-point inputs. Real fixed-point inputs can be either signed or unsigned, while complex fixed-point inputs must be signed. The data type of the maximum values output by the block match the data type of the input. The index values output by the block are `double` when the input is `double`, and `uint32` otherwise.

Value Mode

When **Mode** is set to `Value`, the block computes the maximum value in each column of the M-by-N input matrix u independently at each sample time.

```
val = max(u)      % Equivalent MATLAB code
```

For convenience, length-M 1-D vector inputs and *sample-based* length-M row vector inputs are both treated as M-by-1 column vectors.

The output at each sample time, val, is a 1-by-N vector containing the maximum value of each column in u.

For complex inputs, the block selects the value in each column that has the maximum *magnitude squared* as shown below. For complex value $u = a + bi$, the magnitude squared is $a^2 + b^2$.

The frame status of the output is the same as that of the input.

Index Mode

When **Mode** is set to Index, the block computes the maximum value in each column of the *M*-by-*N* input matrix u,

```
[val,idx] = max(u)      % Equivalent MATLAB code
```
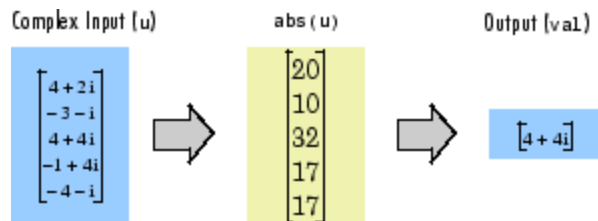
and outputs the sample-based 1-by-*N* index vector, idx. Each value in idx is an integer in the range [1 *M*] indexing the maximum value in the corresponding column of *u*. When inputs to the block are double-precision values, the index values are double-precision values. Otherwise, the index values are 32-bit unsigned integer values.

As in Value mode, length-*M* 1-D vector inputs and *sample-based* length-*M* row vector inputs are both treated as *M*-by-1 column vectors.

When a maximum value occurs more than once in a particular column of *u*, the computed index corresponds to the first occurrence. For example, when the input is the column vector [3 2 1 2 3]', the computed index of the maximum value is 1 rather than 5.

Value and Index Mode

When **Mode** is set to Value and Index, the block outputs both the vector of maxima, val, and the vector of indices, idx.

Running Mode

When **Mode** is set to Running, the block tracks the maximum value of each channel in a *time-sequence* of *M*-by-*N* inputs. For sample-based inputs, the output is a sample-based *M*-by-*N* matrix with each element y_{ij} containing the maximum value observed in element u_{ij} for all inputs

Maximum

since the last reset. For frame-based inputs, the output is a frame-based *M*-by-*N* matrix with each element y_{ij} containing the maximum value observed in the *j*th column of all inputs since the last reset, up to and including element u_{ij} of the current input.

As in the other modes, length-*M* 1-D vector inputs and *sample-based* length-*M* row vector inputs are both treated as *M*-by-1 column vectors.

Resetting the Running Maximum

The block resets the running maximum whenever a reset event is detected at the optional Rst port. The rate of the reset signal must be a positive integer multiple of the rate of the data signal input.

For sample-based inputs, a reset event causes the running maximum for each channel to be initialized to the value in the corresponding channel of the current input. For frame-based inputs, a reset event causes the running maximum for each channel to be initialized to the earliest value in each channel of the current input.

You specify the reset event in the **Reset port** menu:

- None — disables the Rst port.

- Rising edge — Triggers a reset operation when the Rst input does one of the following:

    - Rises from a negative value to a positive value or zero

    - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)

- Falling edge — Triggers a reset operation when the Rst input does one of the following:

  - Falls from a positive value to a negative value or zero

  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero (see the following figure)

- Either edge — Triggers a reset operation when the Rst input is a Rising edge or Falling edge (as described above)

- Non-zero sample — Triggers a reset operation at each sample time that the Rst input is not zero

Maximum

Fixed-Point Data Types

The parameters on the **Fixed-point** pane of the block dialog are only used for complex fixed-point inputs. The sum of the squares of the real and imaginary parts of such an input are formed before a comparison is made, as described in "Value Mode" on page 10-702. The results of the squares of the real and imaginary parts are placed into the product output data type. The result of the sum of the squares is placed into the accumulator data type. These parameters are ignored for other types of inputs.

Examples   The Maximum block in the following model calculates the running maximum of 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 Rst port.
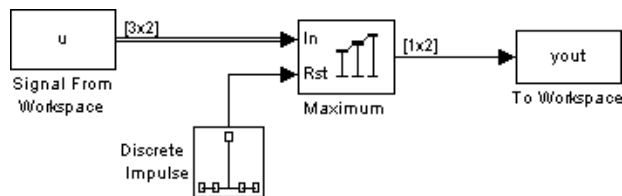
The Maximum block has the following settings:

- **Mode** = Running
- **Reset port** = Non-zero signal

The Signal From Workspace block has the following settings

- **Signal** = u
- **Sample time** = 1/3
- **Samples per frame** = 3

where

```
u
= [6 1 3 -7 2 5 8 0 -1 -3 2 1;1 3 9 2 4 1 6 2 5 0 4 17]'
```

The Discrete Impulse block has the following settings:

- **Delay (samples)** = 2
- **Sample time** = 1
- **Samples per frame** = 1

The block's operation is shown in the figure below.

Maximum
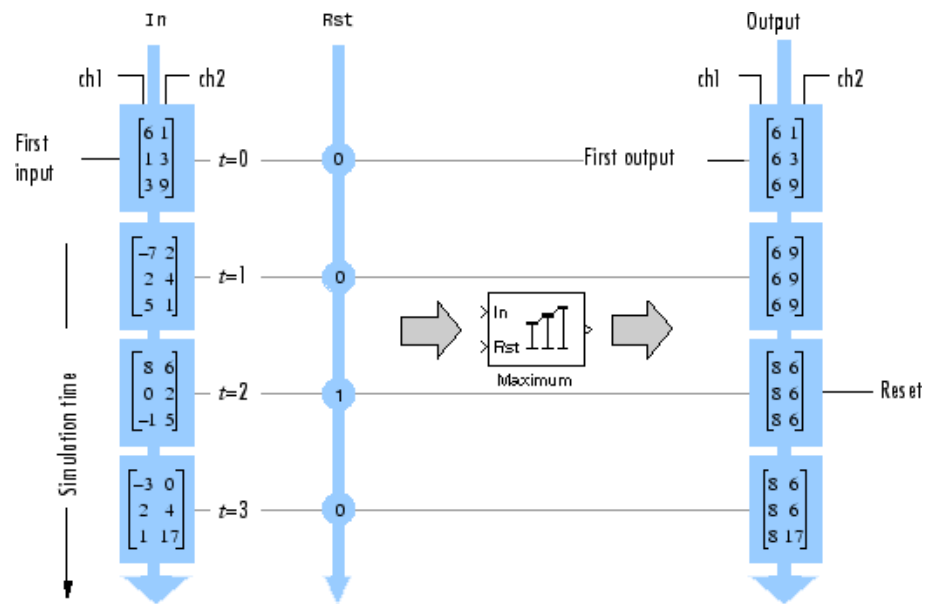
The `statsdem` demo illustrates the operation of several blocks from the Statistics library.

Dialog Box

The **Main** pane of the Maximum block dialog appears as follows:
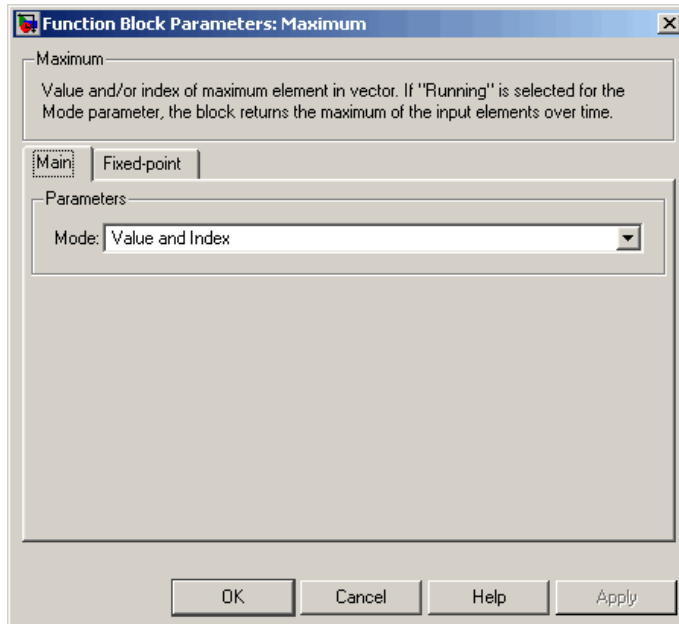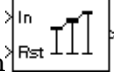
Mode

Specify the block's mode of operation:

- Value — Output the maximum value of each input

- Index — Output the index of the maximum value

- Value and index — Output both the value and the index

- Running — Track the maximum value of the input sequence over time
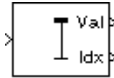
For more information about these modes, refer to "Description" on page 10-702.
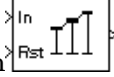
Reset port

Specify the reset event detected at the Rst input port when you select Running for the **Mode** parameter. The rate of the reset signal must be a positive integer multiple of the rate of the data signal input. For information about the possible values of this parameter, refer to "Resetting the Running Maximum" on page 10-704.

The **Fixed-point** pane of the Maximum block dialog appears as follows:

Note The parameters on the **Fixed-point** pane are only used for complex fixed-point inputs. The sum of the squares of the real and imaginary parts of such an input are formed before a comparison is made, as described in "Value Mode" on page 10-702. The results of the squares of the real and imaginary parts are placed into the product output data type. The result of the sum of the squares is placed into the accumulator data type. These parameters are ignored for other types of inputs.

Rounding mode

Select the rounding mode for fixed-point operations.

Overflow mode

Select the overflow mode for fixed-point operations.

Product output

Use this parameter to specify how you would like to designate the product output word and fraction lengths resulting from a complex-complex multiplication in the block. Refer to "Multiplication Data Types" on page 8-16 for more information:

- When you select Same as input, these characteristics will match those of the input to the block.

- When you select Binary point scaling, you are able to enter the word length and the fraction length of the product output, in bits.

- When you select Slope and bias scaling, you are able to enter the word length, in bits, and the slope of the product output. This block requires power-of-two slope and a bias of zero.

Accumulator

Use this parameter to specify the accumulator word and fraction lengths resulting from a complex-complex multiplication in the block. Refer to "Multiplication Data Types" on page 8-16 for more information:

- When you select Same as product output, these characteristics will match those of the product output

- When you select Same as input, these characteristics will match those of the input to the block.

- When you select Binary point scaling, you are able to enter the word length and the fraction length of the accumulator, in bits.

- When you select Slope and bias scaling, you are able to enter the word length, in bits, and the slope of the accumulator. This block requires power-of-two slope and a bias of zero.

Lock scaling against changes by the autoscaling tool
Select this parameter to prevent any fixed-point scaling you specify in this block mask from being overriden by the autoscaling tool in the Fixed-Point Settings interface. For more information about the autoscaling tool, refer to "Fixed-Point Settings Interface" on page 8-28.

Supported Data Types

- Double-precision floating point

- Single-precision floating point

- Fixed point — Signed and unsigned real fixed point, and signed complex fixed-point

- Boolean — The block accepts Boolean inputs to the Rst port.

- 32-bit unsigned integer — When inputs to the block are double-precision values, the index values are double-precision values. Otherwise, the index values are 32-bit unsigned integer values.

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Mean	Signal Processing Blockset
Minimum	Signal Processing Blockset
MinMax	Simulink
`max`	MATLAB

Mean

Purpose   Find mean value of an input or sequence of inputs

Library   Statistics
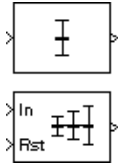
Description The Mean block computes the mean of each column in the input, or
tracks the mean values in a sequence of inputs over a period of time.
The **Running mean** parameter selects between basic operation and
running operation.

The Mean block accepts real and complex fixed-point and floating-point
inputs.

Basic Operation

When you do *not* select the **Running mean** check box, the block
computes the mean of each column of M-by-N input matrix u
independently at each sample time.

```
y = mean(u)      % Equivalent MATLAB code
```

For convenience, length-M 1-D vector inputs and *sample-based*
length-M row vector inputs are both treated as M-by-1 column vectors.

The output at each sample time, y, is a 1-by-N vector containing the
mean value for each column in u. The mean of a complex input is
computed independently for the real and imaginary components, as
shown below.
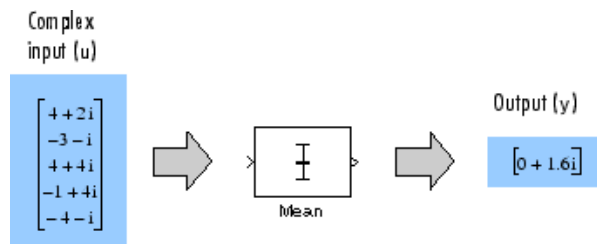
Complex
input (u)

$$\begin{bmatrix} 4 + 2i \\ -3 - i \\ 4 + 4i \\ -1 + 4i \\ -4 - i \end{bmatrix}$$

Output (y)

$$\begin{bmatrix} 0 + 1.6i \end{bmatrix}$$

Mean

The frame status of the output is the same as that of the input.

Running Operation

When you select the **Running mean** check box, the block tracks the mean value of each channel in a *time-sequence* of M-by-N inputs. For sample-based inputs, the output is a sample-based M-by-N matrix with each element y_{ij} containing the mean value of element u_{ij} over all inputs since the last reset. For frame-based inputs, the output is a frame-based M-by-N matrix with each element y_{ij} containing the mean value of the jth column over all inputs since the last reset, up to and including element u_{ij} of the current input.

As in basic operation, length-M 1-D vector inputs and *sample-based* length-M row vector inputs are both treated as M-by-1 column vectors.

Resetting the Running Mean

The block resets the running mean whenever a reset event is detected at the optional Rst port. The rate of the reset signal must be a positive integer multiple of the rate of the data signal input.

When the block is reset for sample-based inputs, the running mean for each channel is initialized to the value in the corresponding channel of the current input. For frame-based inputs, the running mean for each channel is initialized to the earliest value in each channel of the current input.

You specify the reset event by the **Reset port** parameter:

- None disables the Rst port.

- Rising edge — Triggers a reset operation when the Rst input does one of the following:

  - Rises from a negative value to a positive value or zero

  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)

- `Falling edge` — Triggers a reset operation when the `Rst` input does one of the following:

  - Falls from a positive value to a negative value or zero

  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero (see the following figure)

- `Either edge` — Triggers a reset operation when the `Rst` input is a `Rising edge` or `Falling edge` (as described above)

- `Non-zero sample` — Triggers a reset operation at each sample time that the `Rst` input is not zero

> **Note** When running simulations in the Simulink `MultiTasking` mode, reset signals have a one-sample latency. Therefore, when the block detects a reset event, there is a one-sample delay at the reset port rate before the block applies the reset. For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" on page 2-56 and "Models with Multiple Sample Rates" in the Real-Time Workshop User's Guide documentation.

Fixed-Point Data Types

The following diagram shows the data types used within the Mean block for fixed-point signals.
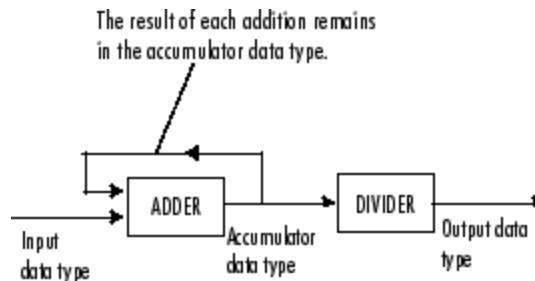
You can set the accumulator and output data types in the block dialog as discussed in "Dialog Box" on page 10-720.

Examples   The Mean block in the following model calculates the running mean of a frame-based 3-by-2 (two-channel) matrix input, u. The running mean is reset at t=2 by an impulse to the block's Rst port.

Mean
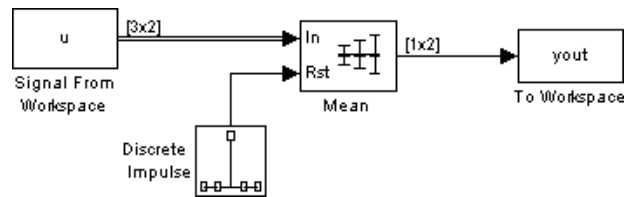
The Mean block has the following settings:

- **Running mean** = Select this check box
- **Reset port** = Non-zero sample

The Signal From Workspace block has the following settings

- **Signal** = u
- **Sample time** = 1/3
- **Samples per frame** = 3

where

```
u = [6 1 3 -7 2 5 8 0 -1 -3 2 1;1 3 9 2 4 1 6 2 5 0 4 17]'
```

The Discrete Impulse block has the following settings:

- **Delay (samples)** = 2
- **Sample time** = 1
- **Samples per frame** = 1

The block's operation is shown in the figure below.
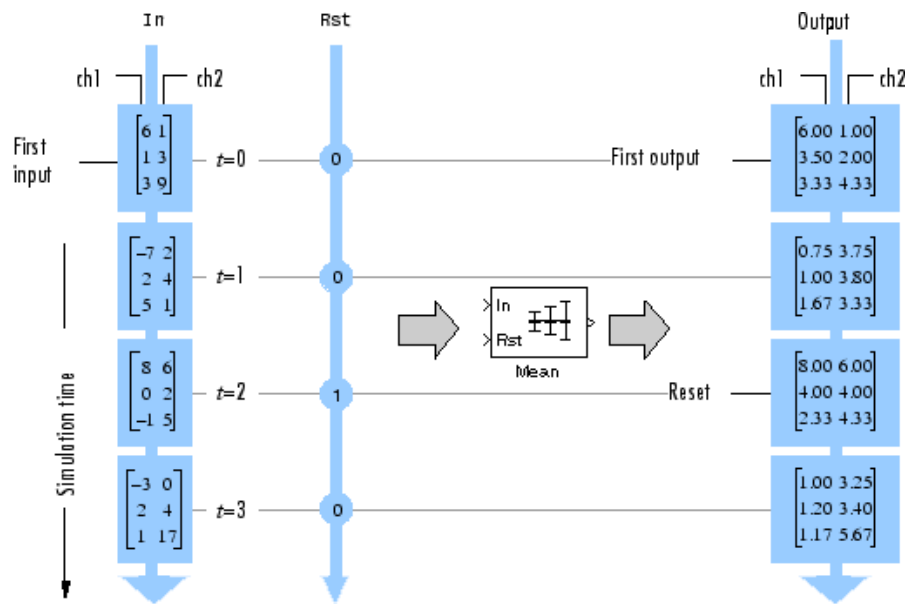
The `statsdem` demo illustrates the operation of several blocks from the Statistics library.

Mean

Dialog Box

The **Main** pane of the Mean block dialog appears as follows.
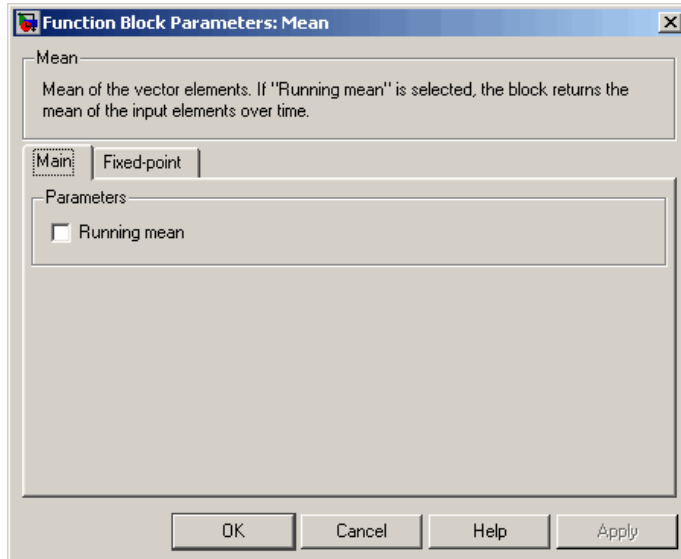
Running mean

Enables running operation when selected.

Reset port

Determines the reset event that causes the block to reset the running mean. The rate of the reset signal must be a positive integer multiple of the rate of the data signal input. This parameter is enabled only when you set the **Running mean** parameter. For more information, see "Resetting the Running Histogram" on page 10-505.

The **Fixed-point** pane of the Mean block dialog appears as follows:

Rounding mode

Select the rounding mode for fixed-point operations.

Overflow mode

Select the overflow mode for fixed-point operations.

Accumulator

Use this parameter to specify the accumulator word and fraction lengths:

- When you select Same as input, these characteristics will match those of the input to the block.

- When you select Binary point scaling, you are able to enter the word length and the fraction length of the accumulator, in bits.

- When you select Slope and bias scaling, you are able to enter the word length, in bits, and the slope of the accumulator. This block requires power-of-two slope and a bias of zero.

Mean

Output
Choose how you will specify the output word length and fraction length:

- When you select Same as accumulator, these characteristics will match those of the accumulator.

- When you select Same as input, these characteristics will match those of the input to the block.

- When you select Binary point scaling, you are able to enter the word length and the fraction length of the output, in bits.

- When you select Slope and bias scaling, you are able to enter the word length, in bits, and the slope of the output. This block requires power-of-two slope and a bias of zero.

Lock scaling against changes by the autoscaling tool
Select this parameter to prevent any fixed-point scaling you specify in this block mask from being overriden by the autoscaling tool in the Fixed-Point Settings interface. For more information about the autoscaling tool, refer to "Fixed-Point Settings Interface" on page 8-28.

Supported Data Types

- Double-precision floating point

- Single-precision floating point

- Fixed point (signed only)

- Boolean — The block accepts Boolean inputs to the Rst port

- 8-, 16-, and 32-bit signed integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Maximum	Signal Processing Blockset
Median	Signal Processing Blockset
Minimum	Signal Processing Blockset
Standard Deviation	Signal Processing Blockset
`mean`	MATLAB

Median

Purpose   Find median value of an input

Library   Statistics

Description   The Median block computes the median value of each column in an
M-by-*N* input matrix.
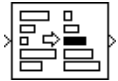
```
y = median(u)     % Equivalent MATLAB code
```

For convenience, length-*M* 1-D vector inputs and *sample-based*
length-*M* row vector inputs are both treated as *M*-by-1 column vectors.

The output at each sample time, *y*, is a sample-based 1-by-*N* vector
containing the median value for each column in *u*.

When *M* is odd, the block sorts the column elements by value, and
outputs the central row of the sorted matrix.

```
s = sort(u); y = s((M+1)/2,:)
```

When *M* is even, the block sorts the column elements by value, and
outputs the average of the two central rows in the sorted matrix.

```
s = sort(u);
y = mean([s(M/2,:);s(M/2+1,:)])
```

Complex inputs are sorted by *magnitude squared*. For complex value
$u = a + bi$, the magnitude squared is $a^2 + b^2$.

The Median block accepts real and complex fixed-point and
floating-point inputs.

Fixed-Point Data Types
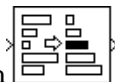
For fixed-point inputs, you can specify accumulator, product output, and
output data types as discussed in "Dialog Box" on page 10-726. Not all
these fixed-point parameters are applicable for all types of fixed-point
inputs. The following table shows when each kind of data type and
scaling is used.

	Output data type	Accumulator data type	Product output data type
Even *M*	X	X	
Odd *M*	X		
Odd *M* and complex	X	X	X
Even *M* and complex	X	X	X

The accumulator and output data types and scalings are used for fixed-point signals when *M* is even. The result of the sum performed while calculating the average of the two central rows of the input matrix is stored in the accumulator data type and scaling. The total result of the average is then put into the output data type and scaling.

The accumulator and product output parameters are used for complex fixed-point inputs. The sum of the squares of the real and imaginary parts of such an input are formed before the input elements are sorted, as described in "Description " on page 10-724. The results of the squares of the real and imaginary parts are placed into the product output data type and scaling. The result of the sum of the squares is placed into the accumulator data type and scaling.
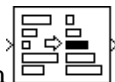
For fixed-point inputs that are both complex and have even *M*, the data types are used in all of the ways described. Therefore, in such cases the accumulator type is used in two different ways.

Median

Dialog Box

The **Main** pane of the Median block dialog appears as follows:
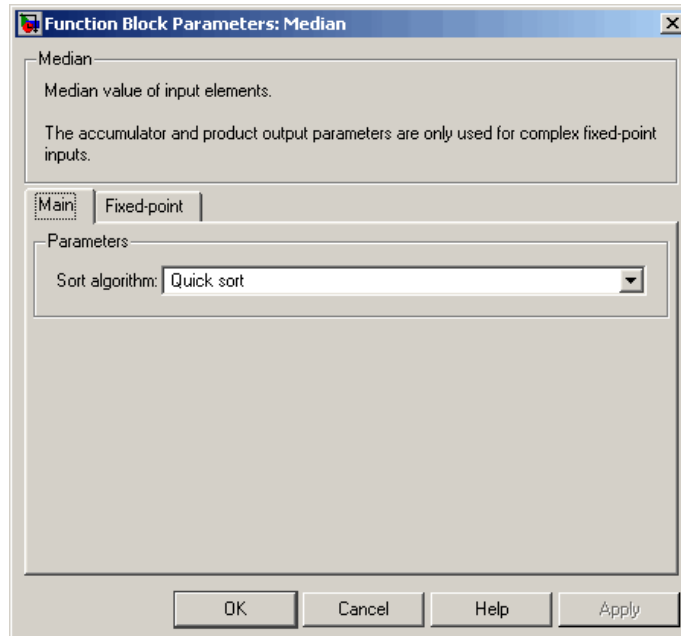
Sort algorithm

Specify whether the elements of the input are sorted using a Quick sort or an Insertion sort algorithm.

The **Fixed-point** pane of the Median block dialog appears as follows:
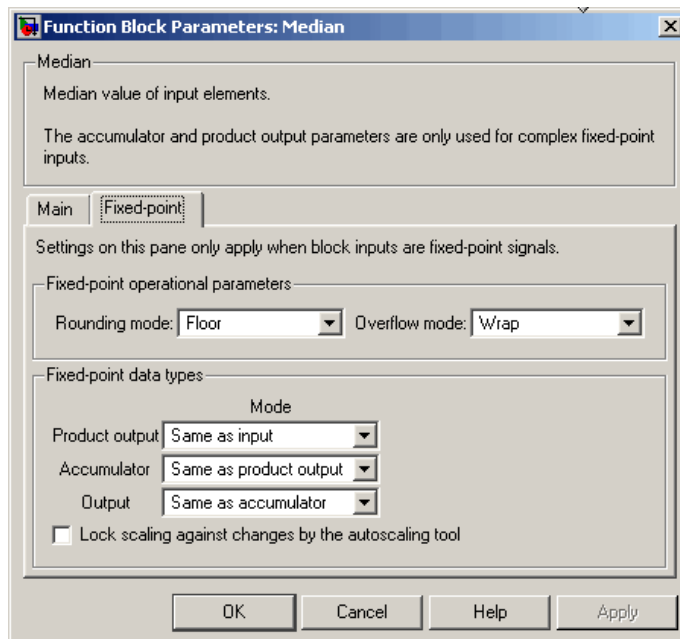
Rounding mode

> Select the rounding mode for fixed-point operations.

Overflow mode

> Select the overflow mode for fixed-point operations.

Note The product output, accumulator, and output parameters listed below are only used in certain cases. Refer to "Fixed-Point Data Types" on page 10-724 for more information.

Product output

> Use this parameter to specify how you would like to designate the product output word and fraction lengths:

- When you select `Same as input`, these characteristics will match those of the input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the product output, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the product output. This block requires power-of-two slope and a bias of zero.

Accumulator

Use this parameter to specify the accumulator word and fraction lengths resulting from a complex-complex multiplication in the block:

- When you select `Same as product output`, these characteristics will match those of the product output

- When you select `Same as input`, these characteristics will match those of the input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the accumulator, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the accumulator. This block requires power-of-two slope and a bias of zero.

Output

Choose how you will specify the output word length and fraction length:

- When you select `Same as accumulator`, these characteristics will match those of the accumulator.

- When you select `Same as product output`, these characteristics will match those of the product output.

- When you select Same as input, these characteristics will match those of the input to the block.

- When you select Binary point scaling, you are able to enter the word length and the fraction length of the output, in bits.

- When you select Slope and bias scaling, you are able to enter the word length, in bits, and the slope of the output. This block requires power-of-two slope and a bias of zero.

Lock scaling against changes by the autoscaling tool
Select this parameter to prevent any fixed-point scaling you specify in this block mask from being overriden by the autoscaling tool in the Fixed-Point Settings interface. For more information about the autoscaling tool, refer to "Fixed-Point Settings Interface" on page 8-28.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• 8-, 16-, 32-, and 128-bit signed integers
	• 8-, 16-, 32-, and 128-bit unsigned integers
Output	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• 8-, 16-, 32-, and 128-bit signed integers
	• 8-, 16-, 32-, and 128-bit unsigned integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

Median

See Also

Maximum	Signal Processing Blockset
Mean	Signal Processing Blockset
Minimum	Signal Processing Blockset
Sort	Signal Processing Blockset
Standard Deviation	Signal Processing Blockset
Variance	Signal Processing Blockset
median	MATLAB

Purpose   Find minimum values in an input or sequence of inputs

Library   Statistics
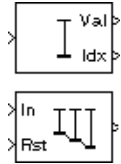
Description   The Minimum block identifies the value and/or position of the smallest element in each column of the input, or tracks the minimum values in a sequence of inputs over a period of time. The **Mode** parameter specifies the block's mode of operation, and can be set to Value, Index, Value and Index, or Running.

The Minimum block supports real and complex floating-point and fixed-point inputs. Fixed-point real inputs can be either signed or unsigned, while fixed-point complex inputs must be signed. The data type of the minimum values output by the block match the data type of the input. The index values output by the block are double when the input is double, and uint32 otherwise.

Value Mode

When **Mode** is set to Value, the block computes the minimum value in each column of the *M*-by-*N* input matrix *u* independently at each sample time.

```
val = min(u)      % Equivalent MATLAB code
```

For convenience, length-*M* 1-D vector inputs and *sample-based* length-*M* row vector inputs are both treated as *M*-by-1 column vectors.

The output at each sample time, val, is a 1-by-*N* vector containing the minimum value of each column in *u*.

For complex inputs, the block selects the value in each column that has the minimum *magnitude squared* as shown below. For complex value $u = a + bi$, the magnitude squared is $a^2 + b^2$.

Minimum
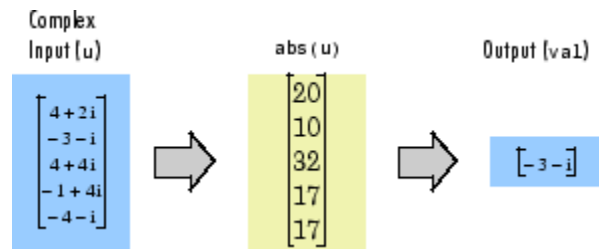
The frame status of the output is the same as that of the input.

Index Mode

When **Mode** is set to `Index`, the block computes the minimum value in each column of the *M*-by-*N* input matrix *u*,

```
[val,idx] = min(u)      % Equivalent MATLAB code
```

and outputs the sample-based 1-by-*N* index vector, `idx`. Each value in `idx` is an integer in the range `[1 M]` indexing the minimum value in the corresponding column of *u*. When inputs to the block are double-precision values, the index values are double-precision values. Otherwise, the index values are 32-bit unsigned integer values.

As in `Value` mode, length-*M* 1-D vector inputs and *sample-based* length-*M* row vector inputs are both treated as *M*-by-1 column vectors.

When a minimum value occurs more than once in a particular column of *u*, the computed index corresponds to the first occurrence. For example, when the input is the column vector `[-1 2 3 2 -1]'`, the computed index of the minimum value is `1` rather than `5`.

Value and Index Mode

When **Mode** is set to `Value and Index`, the block outputs both the vector of minima, `val`, and the vector of indices, `idx`.

Running Mode

When **Mode** is set to `Running`, the block tracks the minimum value of each channel in a *time-sequence* of *M*-by-*N* inputs. For sample-based inputs, the output is a sample-based *M*-by-*N* matrix with each element

y_{ij} containing the minimum value observed in element u_{ij} for all inputs since the last reset. For frame-based inputs, the output is a frame-based *M*-by-*N* matrix with each element y_{ij} containing the minimum value observed in the *j*th column of all inputs since the last reset, up to and including element u_{ij} of the current input.

As in the other modes, length-*M* 1-D vector inputs and *sample-based* length-*M* row vector inputs are both treated as *M*-by-1 column vectors.

Resetting the Running Minimum

The block resets the running minimum whenever a reset event is detected at the optional Rst port. The rate of the reset signal must be a positive integer multiple of the rate of the data signal input.

When the block is reset for sample-based inputs, the running minimum for each channel is initialized to the value in the corresponding channel of the current input. For frame-based inputs, the running minimum for each channel is initialized to the earliest value in each channel of the current input.

Yo specify the reset event by the **Reset port** parameter:

- None disables the Rst port.

- Rising edge — Triggers a reset operation when the Rst input does one of the following:
    - Rises from a negative value to a positive value or zero
    - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)

- `Falling edge` — Triggers a reset operation when the `Rst` input does one of the following:

  - Falls from a positive value to a negative value or zero

  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero (see the following figure)

- `Either edge` — Triggers a reset operation when the `Rst` input is a `Rising edge` or `Falling edge` (as described above)

- `Non-zero sample` — Triggers a reset operation at each sample time that the `Rst` input is not zero

> **Note** When running simulations in the Simulink `MultiTasking` mode, reset signals have a one-sample latency. Therefore, when the block detects a reset event, there is a one-sample delay at the reset port rate before the block applies the reset. For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" on page 2-56 and "Models with Multiple Sample Rates" in the Real-Time Workshop User's Guide documentation.

Fixed-Point Data Types

The parameters on the **Fixed-point** pane of the block dialog are only used for complex fixed-point inputs. The sum of the squares of the real and imaginary parts of such an input are formed before a comparison is made, as described in "Value Mode" on page 10-731. The results of the squares of the real and imaginary parts are placed into the product output data type. The result of the sum of the squares is placed into the accumulator data type. These parameters are ignored for other types of inputs.

Examples   The Minimum block in the following model calculates the running minimum of a frame-based 3-by-2 (two-channel) matrix input. The running minimum is reset at $t=2$ by an impulse to the block's Rst port.
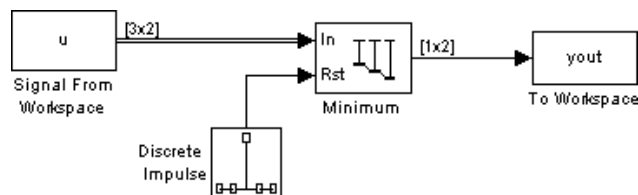
The Minimum block has the following settings:

- **Mode** = Running
- **Reset port** = Non-zero sample

Minimum

The Signal From Workspace block has the following settings

- **Signal** = u
- **Sample time** = 1/3
- **Samples per frame** = 3

where

```
u = [6 1 3 -7 2 5 8 0 -1 -3 2 1;1 3 9 2 4 2 6 2 5 0 4 17]'
```

The Discrete Impulse block has the following settings:

- **Delay (samples)** = 2
- **Sample time** = 1
- **Samples per frame** = 1

The block's operation is shown in the figure below.
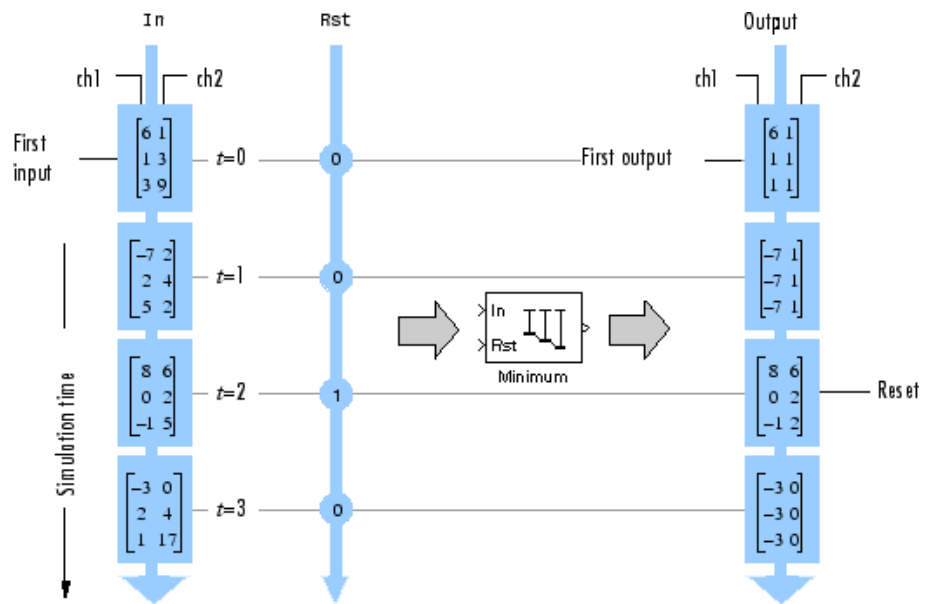
Minimum

Dialog Box

The **Main** pane of the Minimum block dialog appears as follows:
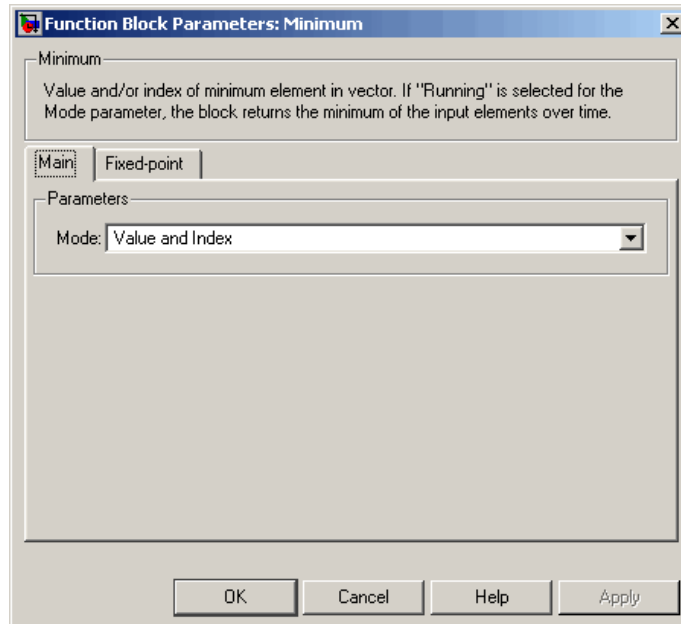
Mode

Specify the block's mode of operation:

- Value — Output the minimum value of each input

- Index — Output the index of the minimum value

- Value and index — Output both the value and the index

- Running — Track the minimum value of the input sequence over time

For more information about these modes, refer to
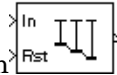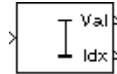
"Description " on page 10-731.

Reset port

Specify the reset event detected at the RST input port when you select Running for the **Mode** parameter. The rate of the reset signal must be a positive integer multiple of the rate of the data signal input. This parameter is enabled only when you set the **Mode** parameter to Running. For information about the possible values of this parameter, see "Resetting the Running Minimum" on page 10-733.

The **Fixed-point** pane of the Minimum block dialog appears as follows:

Minimum

Note The parameters on the **Fixed-point** pane are only used for complex fixed-point inputs. The sum of the squares of the real and imaginary parts of such an input are formed before a comparison is made, as described in "Value Mode" on page 10-731. The results of the squares of the real and imaginary parts are placed into the product output data type. The result of the sum of the squares is placed into the accumulator data type. These parameters are ignored for other types of inputs.

Rounding mode
　　Select the rounding mode for fixed-point operations.

Overflow mode
　　Select the overflow mode for fixed-point operations.

Product output

Use this parameter to specify how you would like to designate the product output word and fraction lengths resulting from a complex-complex multiplication in the block. Refer to "Multiplication Data Types" on page 8-16 for more information:

- When you select `Same as input`, these characteristics will match those of the input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the product output, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the product output. This block requires power-of-two slope and a bias of zero.

Accumulator

Use this parameter to specify the accumulator word and fraction lengths resulting from a complex-complex multiplication in the block. Refer to "Multiplication Data Types" on page 8-16 for more information:

- When you select `Same as product output`, these characteristics will match those of the product output

- When you select `Same as input`, these characteristics will match those of the input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the accumulator, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the accumulator. This block requires power-of-two slope and a bias of zero.

Lock scaling against changes by the autoscaling tool

Select this parameter to prevent any fixed-point scaling you specify in this block mask from being overriden by the autoscaling

tool in the Fixed-Point Settings interface. For more information about the autoscaling tool, refer to "Fixed-Point Settings Interface" on page 8-28.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point — Signed and unsigned real fixed point, and signed complex fixed-point
- Boolean — The block accepts Boolean inputs to the Rst port.
- 32-bit unsigned integer — When inputs to the block are double-precision values, the index values are double-precision values. Otherwise, the index values are 32-bit unsigned integer values.

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Maximum	Signal Processing Blockset
Mean	Signal Processing Blockset
MinMax	Simulink
Histogram	Signal Processing Blockset
min	MATLAB

Purpose   Compute estimate of autoregressive (AR) model parameters using modified covariance method

Library   Estimation / Parametric Estimation
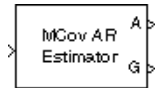
Description   The Modified Covariance AR Estimator block uses the modified covariance method to fit an autoregressive (AR) model to the input data. This method minimizes the forward and backward prediction errors in the least squares sense. The input is a frame of consecutive time samples, which is assumed to be the output of an AR system driven by white noise. The block computes the normalized estimate of the AR system parameters, $A(z)$, independently for each successive input.

$$H(z) = \frac{G}{A(z)} = \frac{G}{1 + a(2)z^{-1} + \ldots + a(p+1)z^{-p}}$$

You specify the order, p, of the all-pole model in the **Estimation order** parameter. To guarantee a valid output, you must set the **Estimation order** parameter to be less than or equal to two thirds the input vector length.

The output port labeled A outputs the normalized estimate of the AR model coefficients in descending powers of z.

    [1 a(2) ... a(p+1)]

The scalar gain, G, is output from the output port labeled G.

See the Burg AR Estimator block reference page for a comparison of the Burg AR Estimator, Covariance AR Estimator, Modified Covariance AR Estimator, and Yule-Walker AR Estimator blocks.

Modified Covariance AR Estimator

**Dialog
Box**
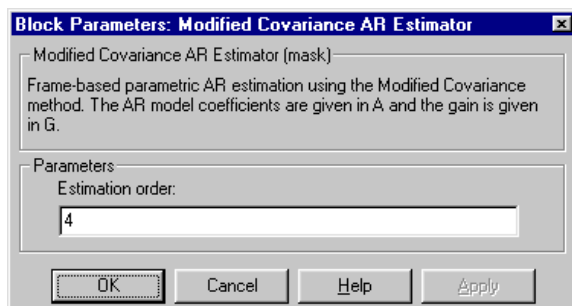
Estimation order
> The order of the AR model, *p*.

References Kay, S. M. *Modern Spectral Estimation: Theory and Application.*
Englewood Cliffs, NJ: Prentice-Hall, 1988.

Marple, S. L., Jr., *Digital Spectral Analysis with Applications.*
Englewood Cliffs, NJ: Prentice-Hall, 1987.

**Supported
Data
Types**

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
A	• Double-precision floating point
	• Single-precision floating point
G	• Double-precision floating point
	• Single-precision floating point

The output data type is the same as the input data type. To learn how
to convert your data types to the above data types in MATLAB and
Simulink, see "Supported Data Types and How to Convert to Them"
on page 7-2.

See Also

Burg AR Estimator	Signal Processing Blockset
Covariance AR Estimator	Signal Processing Blockset
Modified Covariance Method	Signal Processing Blockset
Yule-Walker AR Estimator	Signal Processing Blockset
armcov	Signal Processing Toolbox

Modified Covariance Method

Purpose   Compute parametric spectral estimate using modified covariance method

Library   Estimation / Power Spectrum Estimation

Description
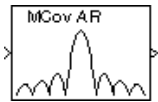
The Modified Covariance Method block estimates the power spectral density (PSD) of the input using the modified covariance method. This method fits an autoregressive (AR) model to the signal by minimizing the forward and backward prediction errors in the least squares sense. The order of the all-pole model is the value specified by the **Estimation order** parameter. To guarantee a valid output, you must set the **Estimation order** parameter to be less than or equal to two thirds the input vector length. The spectrum is computed from the FFT of the estimated AR model parameters.

The input is a sample-based vector (row, column, or 1-D) or frame-based vector (column only) representing a frame of consecutive time samples from a single-channel signal. The block's output (a column vector) is the estimate of the signal's power spectral density at N_{fft} equally spaced frequency points in the range $[0, F_s)$, where F_s is the signal's sample frequency.

When you select **Inherit FFT length from input dimensions**, N_{fft} is specified by the frame size of the input, which must be a power of 2. When you do *not* select **Inherit FFT length from input dimensions**, N_{fft} is specified as a power of 2 by the **FFT length** parameter, and the block zero pads or truncates the input to N_{fft} before computing the FFT. The output is always sample based.

See the Burg Method block reference for a comparison of the Burg Method, Covariance Method, Modified Covariance Method, and Yule-Walker Method blocks.

Examples

The dspsacomp demo compares the modified covariance method with several other spectral estimation methods.

Dialog Box
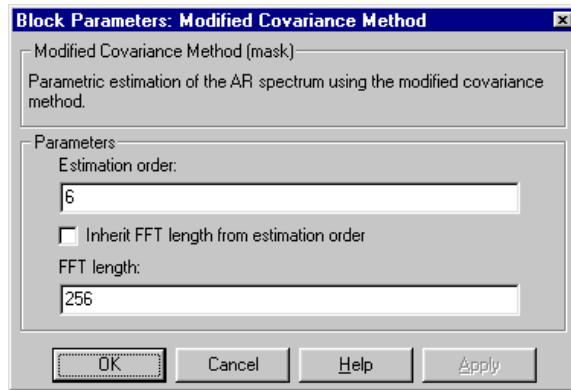
Estimation order
> The order of the AR model.

Inherit FFT length from input dimensions
> When selected, uses the input frame size as the number of data points, N_{fft}, on which to perform the FFT. Tunable.

FFT length
> The number of data points, N_{fft}, on which to perform the FFT. When N_{fft} exceeds the input frame size, the frame is zero-padded as needed. This parameter is enabled when you do not select **Inherit FFT length from input dimensions**.

References
Kay, S. M. *Modern Spectral Estimation: Theory and Application.* Englewood Cliffs, NJ: Prentice-Hall, 1988.

Marple, S. L., Jr., *Digital Spectral Analysis with Applications.* Englewood Cliffs, NJ: Prentice-Hall, 1987.

Modified Covariance Method

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point

The output data type is the same as the input data type. To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Burg Method	Signal Processing Blockset
Covariance Method	Signal Processing Blockset
Modified Covariance AR Estimator	Signal Processing Blockset
Short-Time FFT	Signal Processing Blockset
Yule-Walker Method	Signal Processing Blockset
pmcov	Signal Processing Toolbox

See "Power Spectrum Estimation" on page 6-6 for related information.

Purpose	Generate multiple binary clock signals

Library
- Signal Processing Sources
- Signal Management / Switches and Counters

Description

4-Phase
Clock

The Multiphase Clock block generates a sample-based 1-by-*N* vector of clock signals, where you specify the integer *N* in the **Number of phases** parameter. Each of the *N* phases has the same frequency, *f*, specified in hertz by the **Clock frequency** parameter.

The clock signal indexed by the **Starting phase** parameter is the first to become active, at *t*=0. The other signals in the output vector become active in turn, each one lagging the preceding signal's activation by 1/(*N**f) seconds, the *phase interval*. The period of the sample-based output is therefore 1/(*N**f) seconds.

The *active level* can be either high (1) or low (0), as specified by the **Active level (polarity)** parameter. The duration of the active level, *D*, is set by the **Number of phase intervals over which the clock is active**. This value, which can be an integer value between 1 and *N*-1, specifies the number of phase intervals that each signal should remain in the active state after becoming active. The *active duty cycle* of the signal is *D/N*.

Examples

Configure the Multiphase Clock block in the model below to generate a 100 Hz five-phase output in which the third signal is first to become active. Use a *high* active level with a duration of one interval.
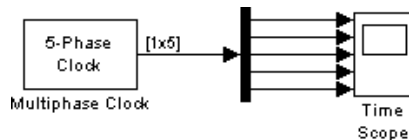
The corresponding settings are as follows:

- **Clock frequency** = 100

Multiphase Clock

- **Number of phases** = 5
- **Starting phase** = 3
- **Number of phase intervals over which the clock is active** = 1
- **Active level (polarity)** = High (1)

The Scope window below shows the Multiphase Clock block's output for these settings. Note that the first active level appears at t=0 on y(3), the second active level appears at t=0.002 on y(4), the third active level appears at t=0.004 on y(5), the fourth active level appears at t=0.006 on y(1), and the fifth active level appears at t=0.008 on y(2). Each signal becomes active 1/(5*100) seconds after the previous signal.
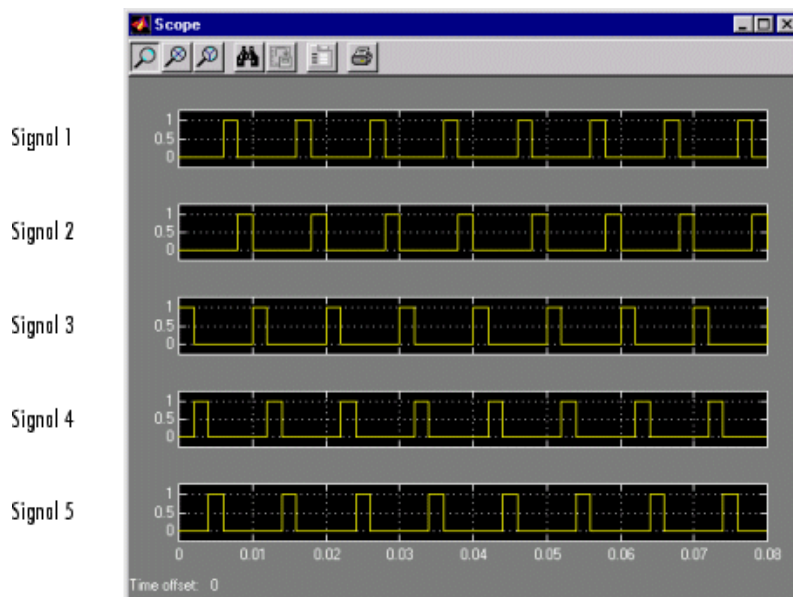
To experiment further, try changing the **Number of phase intervals over which clock is active** setting to 3 so that the active-level duration is three phase intervals (60% duty cycle).

**Dialog
Box**
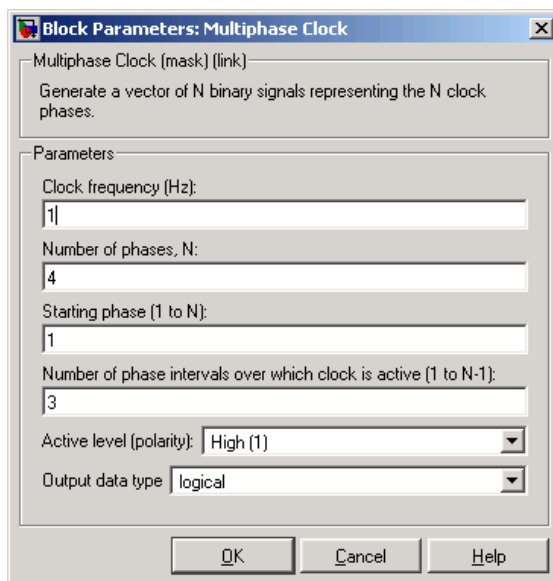
Opening this dialog box causes a running simulation to pause.
See "Changing Source Block Parameters" in the online Simulink
documentation for details.

Clock frequency
    The frequency of all output clock signals.

Number of phases
    The number of different phases, *N*, in the output vector.

Starting phase
    The vector index of the output signal to first become active.
    Tunable.

Number of phase intervals over which clock is active
    The duration of the active level for every output signal. Tunable
    in simulation, but not in Real-Time Workshop external mode.

Active level
    The active level, High (1) or Low (0). Tunable.

Multiphase Clock

Output data type

> The output data type. For information on the `Logical` and
> `Boolean` options of this parameter, see "Effects of Enabling and
> Disabling Boolean Support" on page 7-17.

Supported Data Types

- Double-precision floating point

- Single-precision floating point

- Fixed point

- Boolean — The block might output Boolean values depending on the
  Output data type parameter setting, as described in "Effects of
  Enabling and Disabling Boolean Support" on page 7-17. To learn how
  to disable Boolean output support, see "Steps to Disabling Boolean
  Support" on page 7-18.

To learn how to convert your data types to the above data types in
MATLAB and Simulink, see "Supported Data Types and How to Convert
to Them" on page 7-2.

See Also

Clock	Simulink
Counter	Signal Processing Blockset
Pulse Generator	Simulink
Event-Count Comparator	Signal Processing Blockset

Purpose   Distribute arbitrary subsets of input rows or columns to multiple output ports

Library   Signal Management / Indexing

Description   The Multiport Selector block extracts multiple subsets of rows or columns from *M*-by-*N* input matrix *u*, and propagates each new submatrix to a distinct output port. A length-*M* 1-D vector input is treated as an *M*-by-1 matrix.

The **Indices to output** parameter is a cell array whose *k*th cell contains a one-dimensional indexing expression specifying the subset of input rows or columns to be propagated to the *k*th output port. The total number of cells in the array determines the number of output ports on the block.

When the **Select** parameter is set to Rows, the specified one-dimensional indices are used to select matrix rows, and all elements on the chosen rows are included. When the **Select** parameter is set to Columns, the specified one-dimensional indices are used to select matrix columns, and all elements on the chosen columns are included. A given input row or column can appear any number of times in any of the outputs, or not at all.

When an index references a nonexistent row or column of the input, the block reacts with the behavior specified by the **Invalid index** parameter. The following options are available:

- Clip index — Clip the index to the nearest valid value, and *do not* issue an alert.

  Example: For a 64-by-4 input with **Select** = Rows, an index of 72 is clipped to 64; with **Select** = Columns, an index of 72 is clipped to 4. In both cases, an index of -2 is clipped to 1.

- Clip and warn — Display a warning message in the MATLAB Command Window, and clip as above.

- Generate error — Display an error dialog box and terminate the simulation.

Multiport Selector

Examples

Consider the following **Indices to output** cell array:

```
{4,[1:2
5],[7;8],10:-1:6}
```

This is a four-cell array, which requires the block to generate four independent outputs (each at a distinct port). The table below shows the dimensions of these outputs when **Select** = Rows and the input dimension is *M*-by-*N*.

Cell	Expression	Description	Output Size
1	4	Row 4 of input	1-by-*N*
2	[1:2 5]	Rows 1, 2, and 5 of input	3-by-*N*
3	[7;8]	Rows 7 and 8 of input	2-by-*N*
4	10:-1:6	Rows 10, 9, 8, 7, and 6 of input	5-by-*N*

Dialog Box
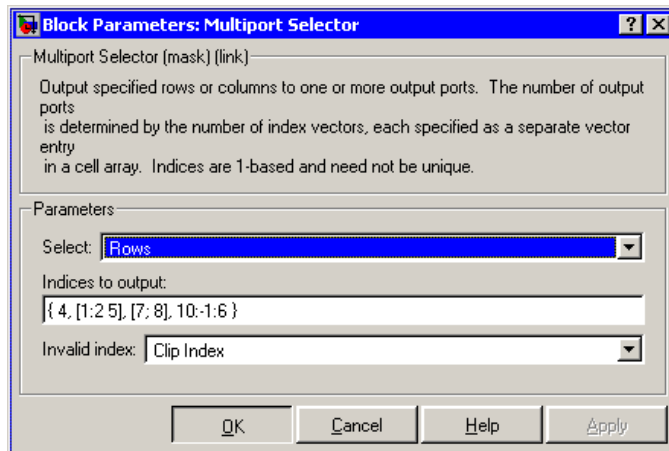
Select

The dimension of the input to select, Rows or Columns.

Indices to output

A cell array specifying the row- or column-subsets to propagate to each of the output ports. The number of cells in the array determines the number of output ports on the block.

Invalid index

Response to an invalid index value.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Custom data types
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
Outputs	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Custom data types
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

Multiport Selector

See Also

Permute Matrix	Signal Processing Blockset
Selector	Simulink
Submatrix	Signal Processing Blockset
Variable Selector	Signal Processing Blockset

Purpose   Output ones or zeros for specified number of sample times

Library   • Signal Processing Sources

            • Signal Management / Switches and Counters

Description The N-Sample Enable block outputs the *inactive* value (0 or 1, whichever is *not* selected in the **Active level** parameter) during the first *N* sample times, where *N* is the **Trigger count** value. Beginning with output sample *N*+1, the block outputs the *active* value (1 or 0, whichever you select in the **Active level** parameter) until a reset event occurs or the simulation terminates.

The output is always sample based.

The **Reset input** check box enables the Rst input port. At any time during the count, a trigger event at the input port resets the counter to its initial state. The rate of the reset signal must be a positive integer multiple of the rate of the data signal input. This block supports triggered subsystems when you select the **Reset input** check box.

You specify the triggering event in the **Trigger type** pop-up menu:

• Rising edge — Triggers a reset operation when the Rst input does one of the following:

     ▪ Rises from a negative value to a positive value or zero

     ▪ Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)

- `Falling edge` — Triggers a reset operation when the `Rst` input does one of the following:

  - Falls from a positive value to a negative value or zero

  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero (see the following figure)

- `Either edge` — Triggers a reset operation when the `Rst` input is a `Rising edge` or `Falling edge` (as described above).

- `Non-zero sample` — Triggers a reset operation at each sample time that the `Rst` input is not zero.

> **Note** When running simulations in the Simulink `MultiTasking` mode, sample-based reset signals have a one-sample latency, and frame-based reset signals have one frame of latency. Thus, there is a one-sample or one-frame delay between the time the block detects a reset event, and when it applies the reset. For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" on page 2-56 and "Models with Multiple Sample Rates" in the Real-Time Workshop User's Guide documentation.

Dialog Box

Opening this dialog box causes a running simulation to pause. See "Changing Source Block Parameters" in the online Simulink documentation for details.

Trigger count

The number of samples for which the block outputs the active value. Tunable.

N-Sample Enable

Active level
    The value to output after the first N sample times, `0` or `1`. Tunable.

Reset input
    Enables the `Rst` input port. The rate of the reset signal must be a positive integer multiple of the rate of the data signal input.

Trigger type
    The type of event that triggers a reset when the `Rst` port is enabled. Nontunable.

Sample time
    The sample period, T_s, for the block's counter. The block switches from the active value to the inactive value at $t=T_s*(N+1)$.

Output data type
    The output data type. Nontunable. For information on the `Logical` and `Boolean` options of this parameter, see "Effects of Enabling and Disabling Boolean Support" on page 7-17.

Supported Data Types

- Double-precision floating point

- Boolean — The block accepts Boolean inputs to the `Rst` port, which is enabled when you set the **Reset input** parameter. The block might output Boolean values depending on the **Output data type** parameter setting, as described in "Effects of Enabling and Disabling Boolean Support" on page 7-17. To learn how to disable Boolean output support, see "Steps to Disabling Boolean Support" on page 7-18.

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Counter	Signal Processing Blockset
N-Sample Switch	Signal Processing Blockset

Purpose   Switch between two inputs after specified number of sample periods

Library   Signal Management / Switches and Counters

Description   The N-Sample Switch block outputs the signal connected to the top input port during the first *N* sample times after the simulation begins or the block is reset, where you specify *N* in the **Switch count** parameter. Beginning with output sample *N*+1, the block outputs the signal connected to the bottom input until the next reset event or the end of the simulation.

You specify the sample period of the output in the **Sample time** parameter (that is, the output sample period is not inherited from the sample period of either input). The block applies a zero-order hold at the input ports, so the value the block reads from a given port between input sample times is the value of the most recent input to that port.

Both inputs must have the same dimension, except in the following two cases:

- When one input is a scalar, the block expands the scalar input to match the size of the other input.

- When one input is a 1-D vector and the other input is a row or column vector with the same number of elements, the block reshapes the 1-D vector to match the dimension of the other input.

The inputs must either both be frame based or both be sample based.

The **Reset input** check box enables the Rst input port. At any time during the count, a trigger event at the Rst port resets the counter to zero. The rate of the reset signal must be a positive integer multiple of the rate of the data signal input. This block supports triggered subsystems when you select the **Reset input** check box.

You specify the triggering event in the **Trigger type** pop-up menu, and can be one of the following:

N-Sample Switch

- `Rising edge` — Triggers a reset operation when the `Rst` input does one of the following:

  - Rises from a negative value to a positive value or zero

  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)

- `Falling edge` — Triggers a reset operation when the `Rst` input does one of the following:

  - Falls from a positive value to a negative value or zero

  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero (see the following figure)

- `Either edge` — Triggers a reset operation when the `Rst` input is a `Rising edge` or `Falling edge` (as described above).

- Non-zero sample — Triggers a reset operation at each sample time that the Rst input is not zero.

Note When running simulations in the Simulink MultiTasking mode, sample-based reset signals have a one-sample latency, and frame-based reset signals have one frame of latency. Thus, there is a one-sample or one-frame delay between the time the block detects a reset event, and when it applies the reset. For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" on page 2-56 and "Models with Multiple Sample Rates" in the Real-Time Workshop User's Guide documentation.

Dialog Box

Switch count

> The number of sample periods, *N*, for which the output is connected to the top input before switching to the bottom input. Tunable.

Reset input

Enables the Rst input port when selected. The rate of the reset signal must be a positive integer multiple of the rate of the data signal input.

Trigger type

The type of event at the Rst port that resets the block's counter. This parameter is enabled when you select **Reset input**. Tunable.

Sample time

The sample period, T_s, for the block's counter. The block switches inputs at $t=T_s*(N+1)$.

Supported Data Types

- Double-precision floating point

- Single-precision floating point

- Fixed point (signed and unsigned)

- Custom data types

- Boolean — The block accepts Boolean inputs to the Rst port, which is enabled when you set the **Reset input** parameter.

- 8-, 16-, and 32-bit signed integers

- 8-, 16-, and 32-bit unsigned integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Counter	Signal Processing Blockset
N-Sample Enable	Signal Processing Blockset

Purpose   Generate real or complex sinusoidal signals

Library   Signal Operations

Description   The NCO block generates a multichannel real or complex sinusoidal
signal, with independent frequency and phase in each output channel.
The amplitude of the created signal is always 1. The block implements
the algorithm as shown in the following diagram:
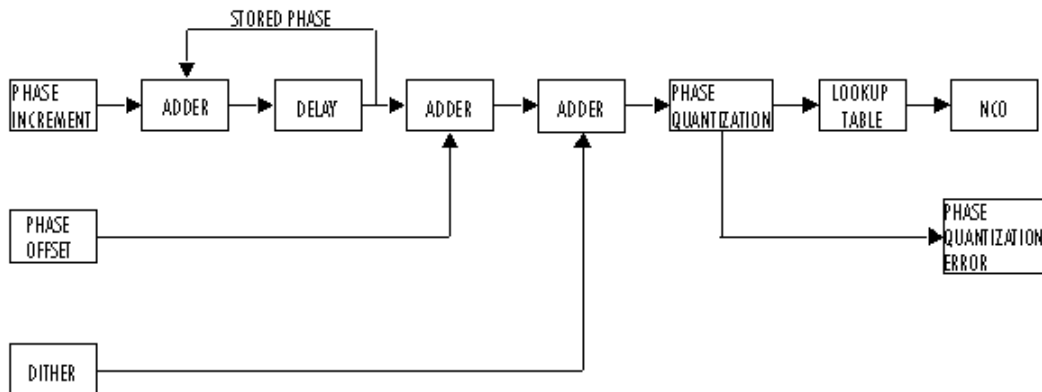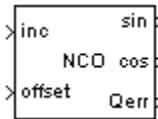
The implementation of a numerically controlled oscillator (NCO) has
two distinct parts. First, a phase accumulator accumulates the phase
increment and adds in the phase offset. In this stage, an optional
internal dither signal can also be added. The NCO output is then
calculated by quantizing the results of the phase accumulator section
and using them to select values from a lookup table.

Given a desired output frequency F_0, calculate the value of the **Phase
increment** block parameter with

$$phase\ increment = (\frac{F_0 \cdot 2^N}{F_s})$$

where N is the accumulator word length and

$$F_s = \frac{1}{T_s} = \frac{1}{sample\ time}$$

The frequency resolution of an NCO is defined by

$$\Delta f = \frac{1}{T_s \cdot 2^N}\ \text{Hz}$$

Given a desired phase offset (in radians), calculate the **Phase offset** block parameter with

$$phase\ offset = \frac{2^N \cdot desired\ phase\ offset}{2\pi}$$

The spurious free dynamic range (SFDR) is estimated as follows for a lookup table with 2^P entries, where P is the number of quantized accumulator bits:

$$SFDR = (6P)\ \text{dB} \qquad \text{without dither}$$
$$SFDR = (6P + 12)\ \text{dB} \quad \text{with dither}$$

The NCO block supports real inputs only. All outputs are real except for the output signal in `Complex exponential` mode.

Fixed-Point Data Types

The following diagram shows the data types used within the NCO block.

I = Integer
A = Accumulator data type
D = Dither bits
Q = Quantized accumulator bits
P = Phase quantization data type
O = Output data type
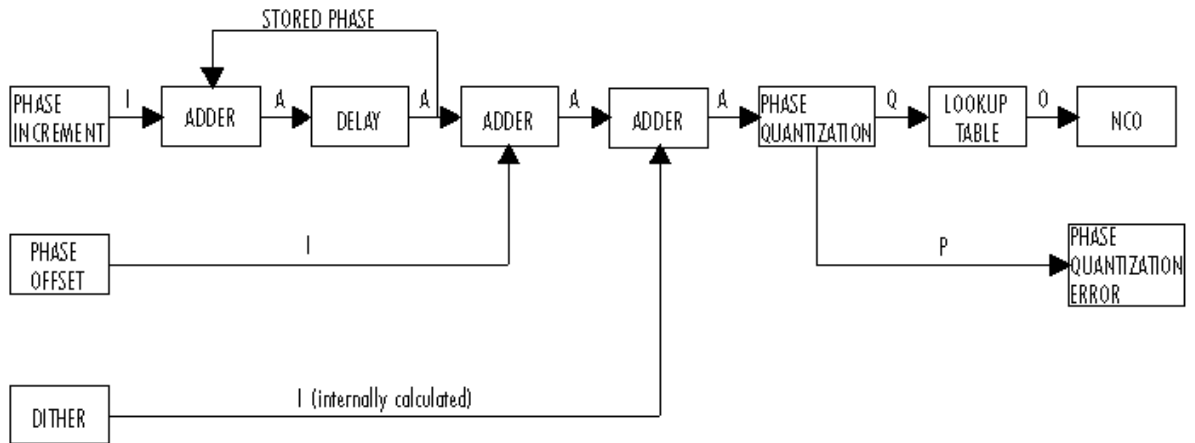
- You can set the accumulator and output data types in the block dialog as discussed in "Dialog Box" on page 10-774 below.

- The phase increment and phase offset inputs must be integers or fixed-point data types with zero fraction length.

- You specify the number of quantized accumulator bits in the **Number of quantized accumulator bits** parameter.

- The phase quantization error word length is equal to the accumulator word length minus the number of quantized accumulator bits, and the fraction length is zero.

Examples   Design an NCO source with the following specifications:

- Desired output frequency $F_0 = 510$ Hz

- Frequency resolution $\Delta f = 0.05$ Hz

- Sample period $T_s = 1/8000$ s

- Desired phase offset $\pi/2$

1 Calculate the number of required accumulator bits from the equation for frequency resolution:

   Note that N must be an integer value. The value of N is rounded up to the nearest integer; 18 accumulator bits are needed to accommodate the value of the frequency resolution.

2 Using this best value of N, calculate the frequency resolution that will be achieved by the NCO block:

$$\Delta f = \frac{1}{T_s \cdot 2^N}\, \text{Hz}$$

$$\Delta f = \frac{1}{\dfrac{1}{8000} \cdot 2^{18}}\ \text{Hz}$$

$$\Delta f = 0.0305$$

3 Calculate the number of quantized accumulator bits from the equation for spurious free dynamic range and the fact that for a lookup table with 2^P entries, P is the number of quantized accumulator bits:

$$SFDR = (6P + 12)\,\text{dB}$$
$$96\ \text{dB} = (6P + 12)\ \text{dB}$$
$$P = 14$$

4 Select the number of dither bits. In general, a good choice for the number of dither bits is the accumulator word length minus the output word length; in this case 4.

5 Calculate the phase increment:

$$phase\ increment = \text{round}(\frac{F_0 \cdot 2^N}{F_s})$$

$$phase\ increment = \text{round}(\frac{501 \cdot 2^{18}}{8000})$$

$$phase\ increment = 16417$$

6 Calculate the phase offset:

$$phase\ offset = \frac{2^{accumulator\ word\ length} \cdot desired\ phase\ offset}{2\pi}$$
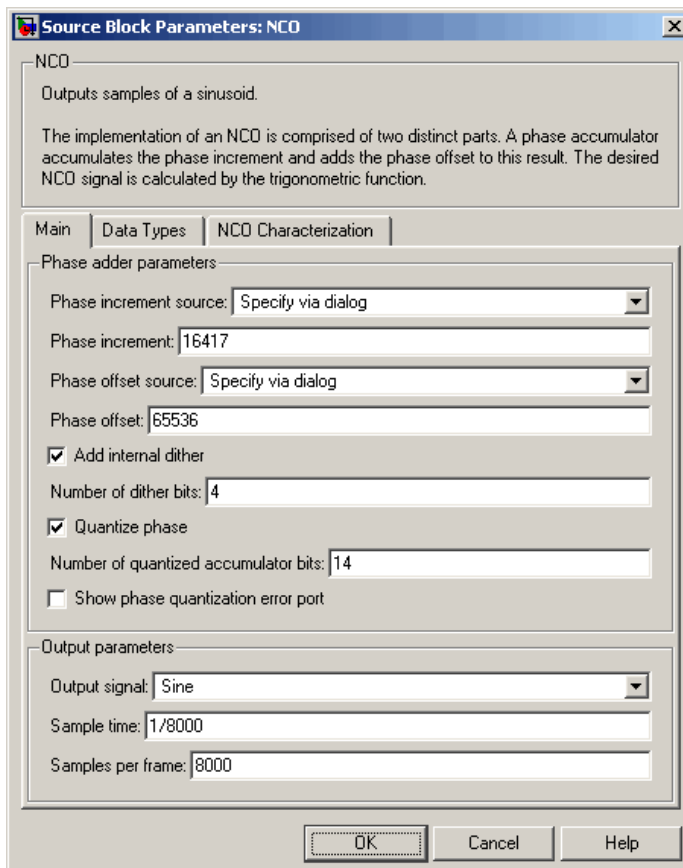
$$phase\ offset = \frac{2^{18} \cdot \dfrac{\pi}{2}}{2\pi}$$

$$phase\ offset = 65536$$

7 Type doc_nco_example at the MATLAB command line to open the following model:
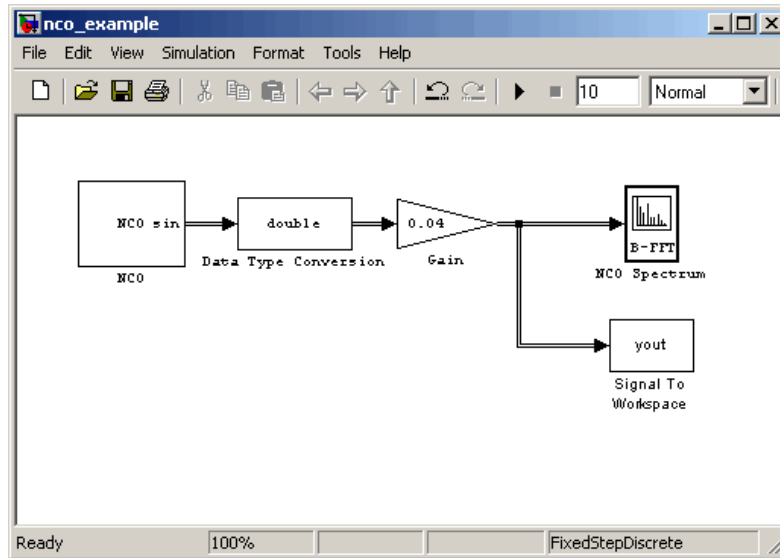
The NCO block in the model is populated with the specifications and quantities you just calculated. The output word length and fraction length depend on the constraints of your hardware; this example uses a word length of 16 and a fraction length of 14. The three panes of the block mask appear as follows:

Source Block Parameters: NCO

NCO

Outputs samples of a sinusoid.

The implementation of an NCO is comprised of two distinct parts. A phase accumulator accumulates the phase increment and adds the phase offset to this result. The desired NCO signal is calculated by the trigonometric function.

Main | Data Types | NCO Characterization

Phase adder parameters

Phase increment source: Specify via dialog

Phase increment: 16417

Phase offset source: Specify via dialog

Phase offset: 65536

☑ Add internal dither

Number of dither bits: 4

☑ Quantize phase

Number of quantized accumulator bits: 14

☐ Show phase quantization error port
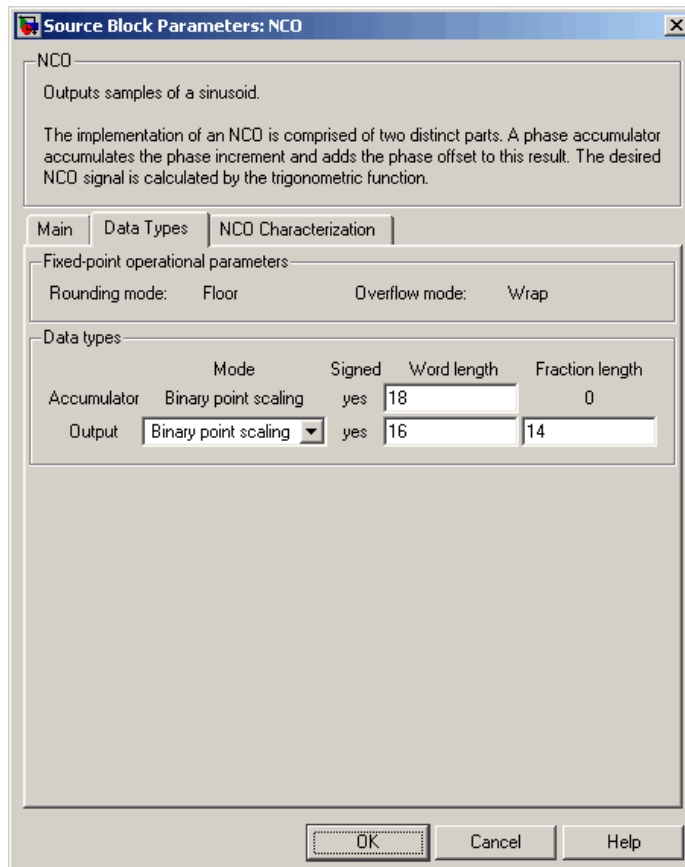
Output parameters

Output signal: Sine

Sample time: 1/8000

Samples per frame: 8000

OK   Cancel   Help
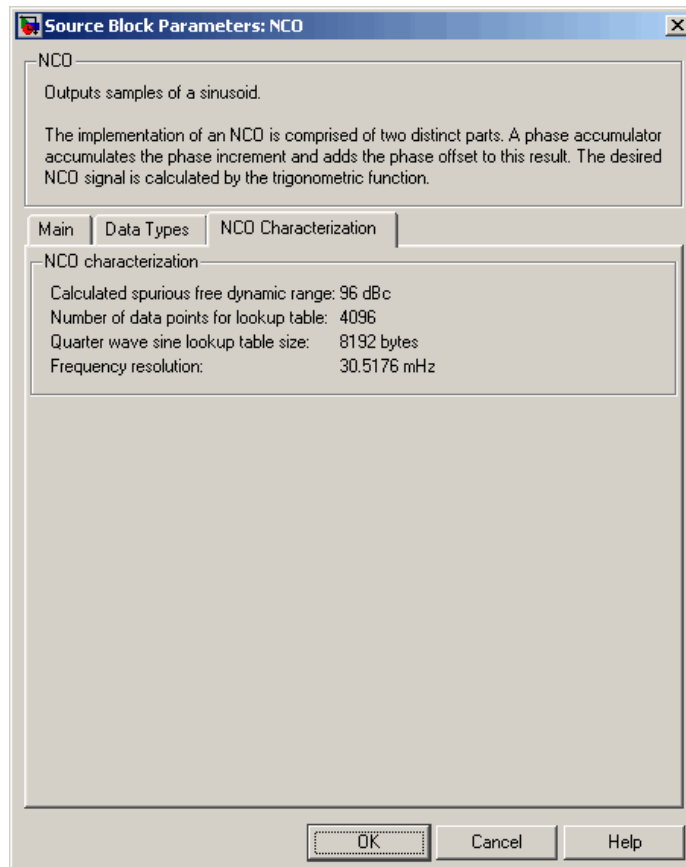
Looking at the **NCO Characterization** pane, you can verify that the specifications of this problem have been met.

8 Experiment with the model to observe the effects on the output shown on the Spectrum Scope. For example, try turning dithering on and off, and try changing the number of dither bits.

Dialog Box

The **Main** pane of the NCO dialog appears as follows:
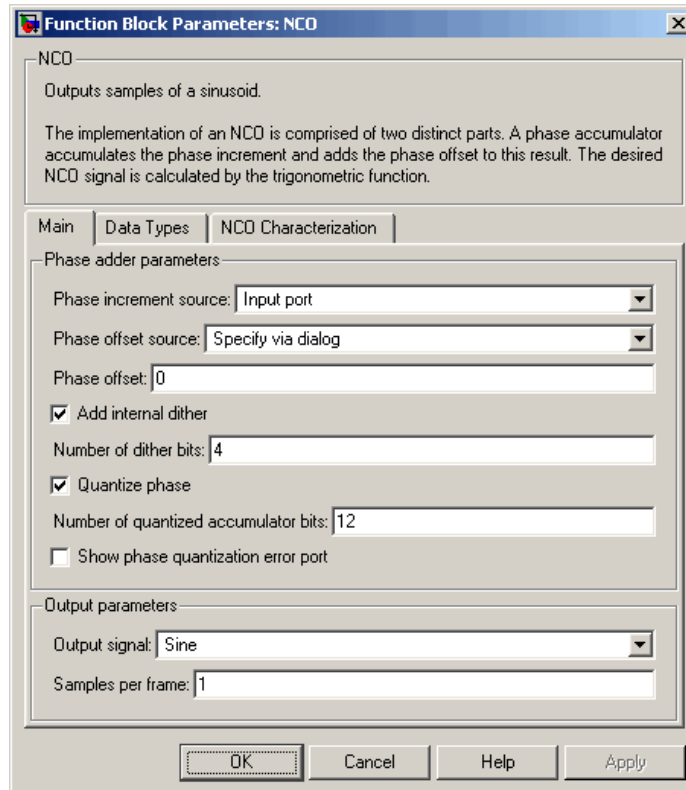
Phase increment source

Choose how you will specify the phase increment. The phase increment can come from an input port or from the dialog.

- If you select Input port, the inc port appears on the block icon.

- If you select Specify via dialog, the **Phase increment** parameter appears.

Phase increment

Specify the phase increment. Only integer data types, including fixed-point data types with zero fraction length, are allowed.

This parameter is visible only if `Specify via dialog` is selected for the **Phase increment source** parameter.

Phase offset source

Choose how you will specify the phase offset. The phase offset can come from an input port or from the dialog.

- If you select `Input port`, the `offset` port appears on the block icon.

- If you select `Specify via dialog`, the **Phase offset** parameter appears.

Phase offset

Specify the phase offset. Only integer data types, including fixed-point data types with fraction length, are allowed.

This parameter is visible only if `Specify via dialog` is selected for the **Phase offset source** parameter.

Add internal dither

Select to add internal dithering to the NCO algorithm. Dithering is added using the PN Sequence Generator from the Communications Blockset.

Number of dither bits

Specify the number of dither bits.

This parameter is visible only if **Add internal dither** is selected.

Quantize phase

Select to enable quantization of the accumulated phase.

Number of quantized accumulator bits

Specify the number of quantized accumulator bits. This determines the number of entries in the lookup table. The number

of quantized accumulator bits must be less than the accumulator word length.

This parameter is visible only if **Quantize phase** is selected.

Show phase quantization error port

Select to output the phase quantization error. When you select this, the Qerr port appears on the block icon.

This parameter is visible only if **Quantize phase** is selected.

Output signal

Choose whether the block should output a Sine, Cosine, Complex exponential, or Sine and cosine signals. If you select Sine and cosine, the two signals will output on different ports.

Sample time

Specify the sample time in seconds when the block is acting as a source. When either the phase increment or phase offset come in via block input ports, the sample time is inherited and this parameter is not visible.

Samples per frame

Specify the number of samples per frame when the number of samples per frame is greater than one. When it exists, the phase offset input port has the same frame status as the output port(s). The phase increment input port, when it exists, does not support frames.

The **Data Types** pane of the NCO dialog appears as follows:
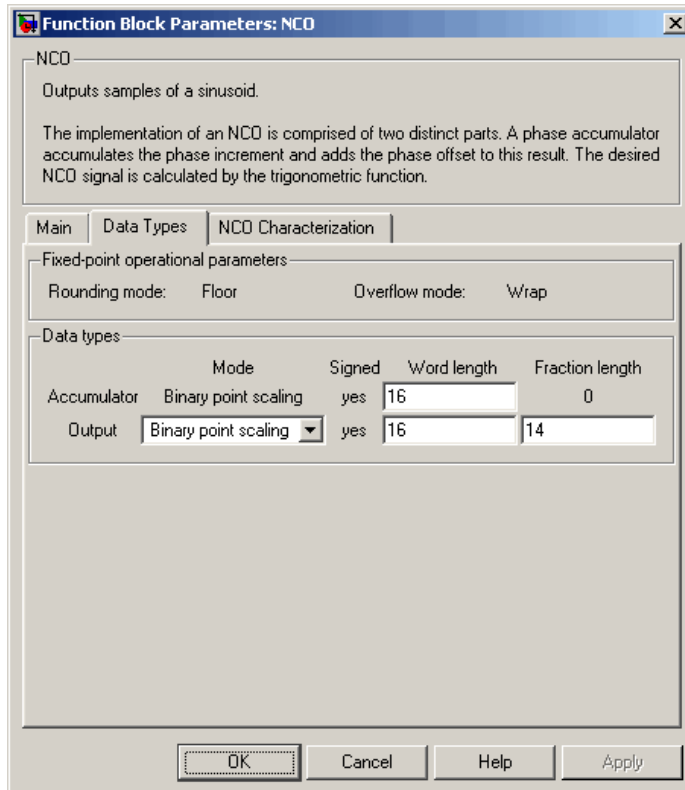
Rounding mode

The rounding mode used for this block when inputs are fixed point is always Floor.

Overflow mode

The overflow mode used for this block when inputs are fixed point is always Wrap.

Accumulator

Specify the word length of the accumulator data type. The fraction length is always zero; this is an integer data type.

Output

Specify the output data type.

- Choose `double` or `single` for a floating-point implementation.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the output, in bits.

The **NCO Characterization** pane of the NCO dialog appears as follows:
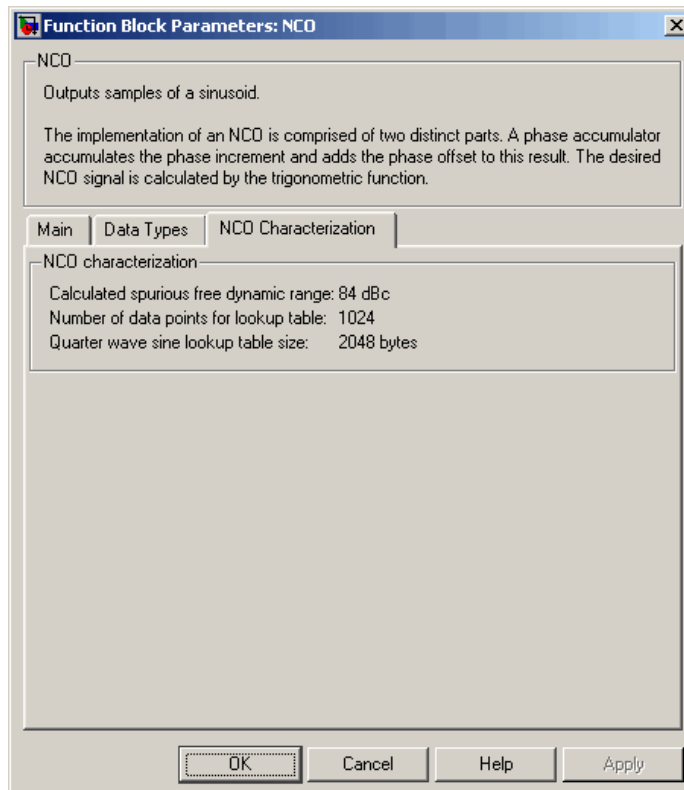
The **NCO Characterization** pane does not have any parameters. Instead, it provides you with details on the NCO signal currently being implemented by the block:

- `Calculated spurious free dynamic range` — The spurious free dynamic range (SFDR) is calculated as follows for a lookup table with 2^P entries:

  $SFDR = (6P)$ dB   without dither

  $SFDR = (6P + 12)$ dB   with dither

- `Number of data points for lookup table` — The lookup table is implemented as a quarter-wave sine table. The number of lookup table data points is defined by

$$2^{\text{number of quantized accumulator bits}-2}$$

- `Quarter wave sine lookup table size` — The quarter wave sine lookup table size is defined by

$$\frac{(\text{number of lookup table data points}) \cdot (\text{output word length})}{8} \text{ bytes}$$

Supported Data Types

Port	Supported Data Types
inc	• Fixed point (signed) with zero fraction length • 8-, 16-, and 32-bit signed integers
offset	• Fixed point (signed) with zero fraction length • 8-, 16-, and 32-bit signed integers
sin	• Double-precision floating point • Single-precision floating point • Fixed point (signed)
Qerr	• 8-, 16-, and 32-bit signed integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

PN Sequence Generator	Communications Blockset
Sine Wave	Signal Processing Blockset

Purpose   Normalize input by its 2-norm or squared 2-norm

Library   Math Functions / Math Operations

Description

$$\frac{u}{\|u\|^2}$$
Normalization

The Normalization block independently normalizes each column of the *M*-by-*N* matrix input, *u*.
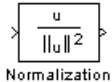
The block accepts the following types of inputs:

- Frame-based vectors and matrices

- Sample-based row and column vectors

- Sample-based unoriented (1-D) vectors

Note the block does not accept sample-based full matrix inputs.

The Normalization block accepts real and complex inputs. The block accepts floating-point signals only for the 2-norm mode, and both fixed-point and floating-point signals for the squared 2-norm mode.

The output *always* has the same dimension and frame status as the input. For convenience, length-*M* 1-D vectors and *sample-based* length-*M* row vectors are both treated as *M*-by-1 column vectors.

2-Norm

The 2-norm mode is supported for floating-point inputs only. When you specify 2-norm for the **Norm** parameter, the block normalizes the *j*th input column as follows

$$y_{ij} = \frac{u_{ij}}{\|u\|_j + b}$$

where you specify *b* in the **Normalization bias** parameter, and $\|u\|_j$ is the 2-norm (or *Euclidean* norm) of the *j*th input column.

$$\|u\|_j = \sqrt{\left|u_{1j}\right|^2 + \left|u_{2j}\right|^2 + \cdots + \left|u_{Mj}\right|^2}$$

Equivalently,

```
y = u ./ (norm(u) + b)          % Equivalent MATLAB code
```

The normalization bias, *b*, is typically chosen to be a small positive constant (for example, 1e-10) that prevents potential division by zero.

Squared 2-Norm

The squared 2-norm mode is supported for both fixed-point and floating-point inputs. When you specify Squared 2-norm for the **Norm** parameter, the block normalizes the *j*th input column as follows

$$y_{ij} = \frac{u_{ij}}{\|u\|_j^2 + b}$$

where

$$\|u\|_j^2 = |u_{1j}|^2 + |u_{2j}|^2 + \cdots + |u_{Mj}|^2$$

Equivalently,

```
y = u ./ (norm(u).^2 + b)          % Equivalent MATLAB code
```

Fixed-Point Data Types

The following diagram shows the data types used within the Normalization block for fixed-point signals (squared 2-norm mode).
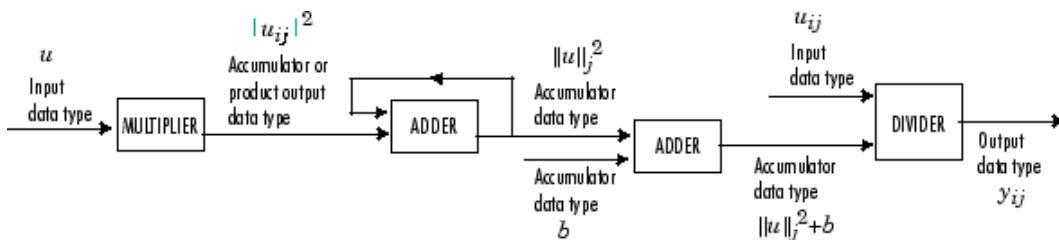
The output of the multiplier is in the product output data type when the input is real. When the input is complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed, refer to "Multiplication Data Types"

on page 8-16. You can set the accumulator, product output, and output data types in the block dialog as discussed in "Dialog Box" on page 10-783 below.

Dialog Box

The **Main** pane of the Normalization dialog appears as follows:
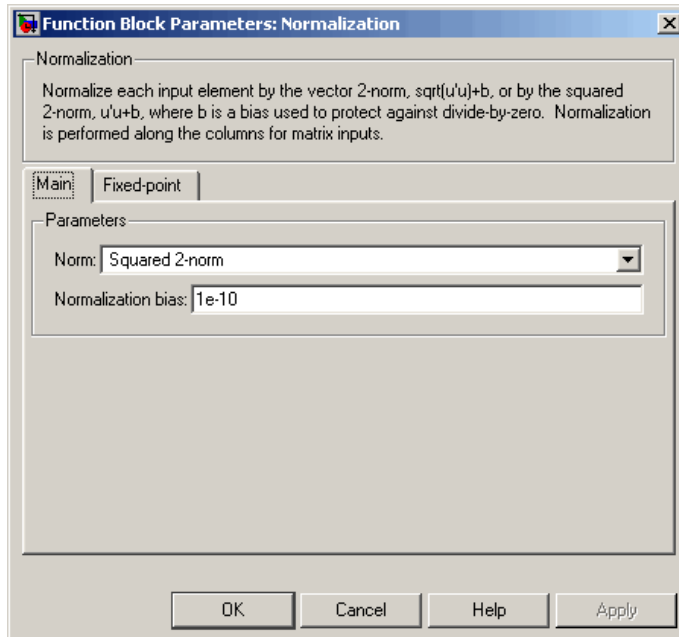
Norm

Specify the type of normalization to apply, 2-norm or Squared 2-norm. 2-norm mode supports floating-point signals. Squared 2-norm supports both fixed-point and floating-point signals. Tunable.

Normalization bias

Specify the real value *b* to be added in the denominator to avoid division by zero. Tunable.

Normalization

The **Fixed-Point** pane of the Normalization dialog appears as follows:
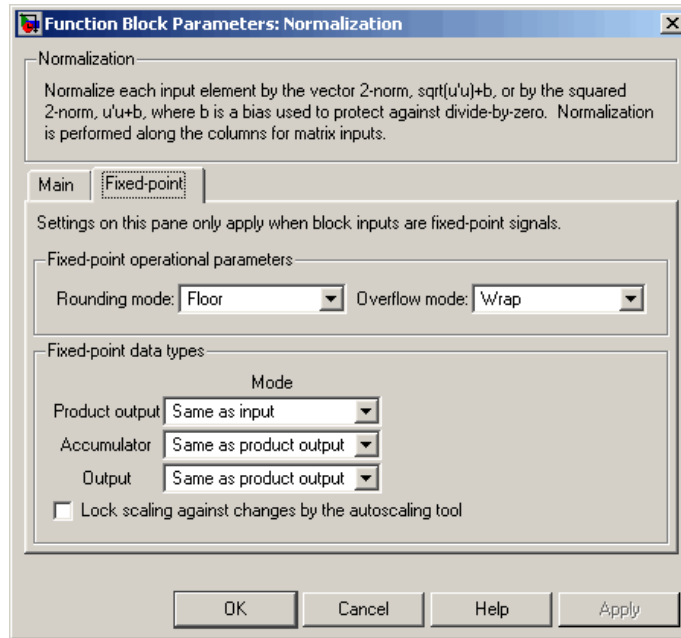
Note The parameters on this pane are only applicable to fixed-point signals when the block is in squared 2-norm mode. Refer to "Fixed-Point Data Types" on page 10-782 for a diagram of how the product output, accumulator, and output data types are used in this case.

Rounding mode
　　Select the rounding mode for fixed-point operations.

Overflow mode
　　Select the overflow mode for fixed-point operations.

Product output

Use this parameter to specify how you would like to designate the product output word and fraction lengths:

- When you select `Same as input`, these characteristics will match those of the input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the product output, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the product output. This block requires power-of-two slope and a bias of zero.

Accumulator

Use this parameter to specify the accumulator word and fraction lengths resulting from a complex-complex multiplication in the block. The bias *b* is also quantized into the accumulator data type:

- When you select `Same as product output`, these characteristics will match those of the product output

- When you select `Same as input`, these characteristics will match those of the input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the accumulator, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the accumulator. This block requires power-of-two slope and a bias of zero.

Output

Choose how you will specify the output word length and fraction length:

- When you select `Same as accumulator`, these characteristics will match those of the accumulator.

Normalization

- When you select `Same as product output`, these characteristics will match those of the product output.

- When you select `Same as input`, these characteristics will match those of the input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the output, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the output. This block requires power-of-two slope and a bias of zero.

Lock scaling against changes by the autoscaling tool
Select this parameter to prevent any fixed-point scaling you specify in this block mask from being overriden by the autoscaling tool in the Fixed-Point Settings interface. For more information about the autoscaling tool, refer to "Fixed-Point Settings Interface" on page 8-28.

Supported Data Types

- Double-precision floating point

- Single-precision floating point

- Fixed point (signed only)

- 8-, 16-, and 32-bit signed integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Matrix Scaling	Signal Processing Blockset
Reciprocal Condition	Signal Processing Blockset
norm	MATLAB

Purpose   Truncate vectors by removing or keeping beginning or ending values

Library   Signal Operations
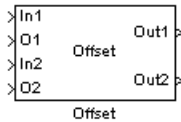
Description   The Offset block removes or keeps values from the beginning or end
of a vector and outputs the result in a vector of user-specified length.
The inputs to the In ports (In1, In2, ...) can be scalars or vectors, but
they must be the same size and data type. The offset values are the
inputs to the O ports (O1, O2, ...); they must be scalar values with the
same data type. These offset values should be integer values because
they determine the number of values the block discards or retains
from each input vector. The block rounds any offset value that is a
noninteger value to the nearest integer value. There is one output port
for each pair of In and O ports. This block supports sample-based and
frame-based signals.

Use the **Mode** parameter to determine which values the block discards
or retains from the input vector. To discard the initial values of the
vector, select Remove beginning samples. To discard the final values
of the vector, select Remove ending samples. To retain the initial
values of the vector, select Keep beginning samples. To retain the
final values of a vector, select Keep ending samples.

Use the **Number of input data-offset pairs** parameter to specify the
number of inputs to the block. The number of input ports is twice the
scalar value you enter. For example, if you enter 3, ports In1, O1, In2,
O2, In3, and O3 appear on the block.

The block uses the **Output port length** parameter to determine the
length of the output vectors. If you select Same as input, the block
outputs vectors that are the same length as the input to the In ports.
If you select User-defined, the **Output length** parameter appears.
Enter a scalar that represents the desired length of the output vectors.
If your desired output length is greater than the number of values you
extracted from your input vector, the block zero-pads the end of the
vector to reach the length you specified.

Use the **Action for out of range offset value** parameter to determine
how the block behaves when an offset value is not in the range

Offset

$0 \le \text{offset value} \le N$, where N is the input vector length. Select `Clip` if you want any offset values less than 0 to be set to 0 and any offset values greater than N to be set to N. Select `Clip and warn` if you want to be warned when any offset values less than 0 are set to 0 and any offset values greater than N are set to N. Select `Error` if you want the simulation to stop and display an error when the offset values are out of range.

Dialog Box
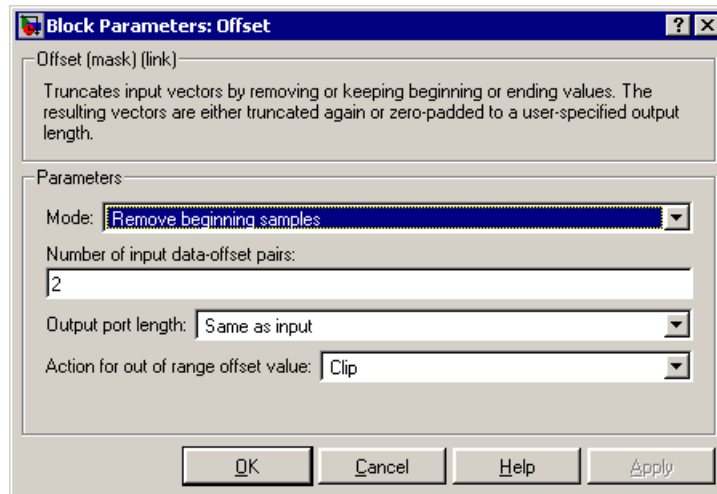
Mode

Use this parameter to determine which values the block discards or retains from the input vector. Your choices are `Remove beginning samples`, `Remove ending samples`, `Keep beginning samples`, and `Keep ending samples`.

Number of input data-offset pairs

Specify the number of inputs to the block. The number of input ports is twice the scalar value you enter.

Output port length

Use this parameter to specify the length of the output vectors. If you select `Same as input`, the output vectors are the same length

as the input vectors. If you select `User-defined`, you can enter the desired length of the output vectors.

Output length

Enter a scalar that represents the desired length of the output vectors. This parameter is visible if, for the **Output port length** parameter, you select `User-defined`.

Action for out of range offset value

Use this parameter to determine how the block behaves when an offset value is not in the range such that $0 \leq \text{offset value} \leq N$, where N is the input vector length. When you want any offset values less than 0 to be set to 0 and any offset values greater than N to be set to N, select `Clip`. When you want to be warned when any offset values less than 0 are set to 0 and any offset values greater than N are set to N, select `Clip and warn`. When you want the simulation to stop and display an error when the offset values are out of range, select `Error`.

Offset

Supported Data Types

Port	Supported Data Types
In	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
O	• Double-precision floating point • Single-precision floating point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Out	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers

Purpose   Implement overlap-add method of frequency-domain filtering

Library   Filtering / Filter Designs

Description   The Overlap-Add FFT Filter block uses an FFT to implement the *overlap-add method*, a technique that combines successive frequency-domain filtered sections of an input sequence.

> Overlap
> Add

Valid inputs to this block are 1-D vectors, sample-based vectors, frame-based vectors, and frame-based full matrices. All outputs are unbuffered into sample-based row vectors. The length of the output vector is equal to the number of channels in the input vector. An M-by-1 *sample-based* input has M channels, so it would result in a length-M sample-based output vector. An M-by-1 *frame-based* input has only one channel, so would result in a 1-by-1 (scalar) output.

The block's data output rate is M times faster than its data input rate, where M is the input frame-size. Thus, the block's data input and output rates are the same when the inputs are 1-D vectors, sample-based vectors, or frame-based row vectors. For frame-based column and frame-based full-matrix inputs, the block's data output rate is M times greater than the block's data input rate.

1-D vectors are treated as length-N sample-based vectors, and result in sample-based length-N row vectors.

The block breaks the scalar input sequence u, of length nu, into length-L nonoverlapping data sections,
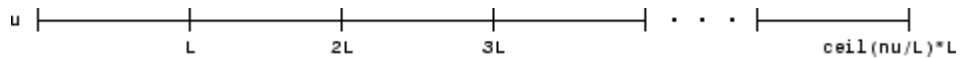
which it linearly convolves with the filter's FIR coefficients,

$$H(z) = B(z) = b_1 + b_2 z^{-1} + \ldots + b_{n+1} z^{-n}$$

The numerator coefficients for H(*z*) are specified as a vector by the **FIR coefficients** parameter. The coefficient vector, b = [b(1) b(2) ... b(n+1)], can be generated by one of the filter

design functions in the Signal Processing Toolbox, such as `fir1`. All filter states are internally initialized to zero.

When either the filter coefficients or the inputs to the block are complex, the **Output** parameter should be set to `Complex`. Otherwise, the default **Output** setting, `Real`, instructs the block to take only the real part of the solution.

The block's overlap-add operation is equivalent to

```
y = ifft(fft(u(i:i+L-1),nfft) .* fft(b,nfft))
```

where you specify `nfft` in the **FFT size** parameter as a power-of-two value greater (typically *much* greater) than n+1. Values for **FFT size** that are not powers of two are rounded upwards to the nearest power-of-two value to obtain `nfft`.

The block overlaps successive output sections by `n` points and sums them.
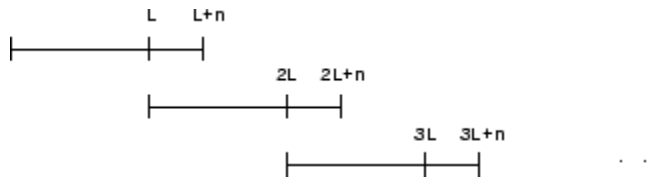
The first `L` samples of each summation are output in sequence. The block chooses the parameter `L` based on the filter order and the FFT size.

```
L = nfft - n
```

Latency

In *single-tasking* operation, the Overlap-Add FFT Filter block has a latency of `nfft-n+1` samples. The first `nfft-n+1` consecutive outputs from the block are zero; the first filtered input value appears at the output as sample `nfft-n+2`.

In *multitasking* operation, the Overlap-Add FFT Filter block has a latency of `2*(nfft-n+1)` samples. The first `2*(nfft-n+1)` consecutive

outputs from the block are zero; the first filtered input value appears at the output as sample `2*(nfft-n)+3`.

Note For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" on page 2-56 and "Models with Multiple Sample Rates" in the Real-Time Workshop User's Guide documentation.

Dialog Box

FFT size
  The size of the FFT, which should be a power-of-two value greater than the length of the specified FIR filter.

FIR coefficients
  The filter numerator coefficients.

Output
  The complexity of the output; Real or Complex. When the input signal or the filter coefficients are complex, this should be set to Complex.

Overlap-Add FFT Filter

References   Oppenheim, A. V. and R. W. Schafer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989.

Proakis, J. and D. Manolakis. *Digital Signal Processing*. 3rd ed. Englewood Cliffs, NJ: Prentice-Hall, 1996.

Supported Data Types
- Double-precision floating point
- Single-precision floating point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Overlap-Save FFT Filter   Signal Processing Blockset

Purpose   Implement overlap-save method of frequency-domain filtering

Library   Filtering / Filter Designs

Description   The Overlap-Save FFT Filter block uses an FFT to implement
the *overlap-save method*, a technique that combines successive
frequency-domain filtered sections of an input sequence.

Valid inputs to this block are 1-D vectors, sample-based vectors,
frame-based vectors, and frame-based full matrices. All outputs are
unbuffered into sample-based row vectors. The length of the output
vector is equal to the number of channels in the input vector. An M-by-1
sample-based input has M channels, so it would result in a length-M
sample-based output vector. An M-by-1 frame-based input has only one
channel, so would result in a 1-by-1 (scalar) output.

The block's data output rate is M times faster than its data input
rate, where M is the input frame-size. Thus, the block's data input
and output rates are the same when the inputs are 1-D vectors,
sample-based vectors, or frame-based row vectors. For frame-based
column and frame-based full-matrix inputs, the block's data output rate
is M times greater than the block's data input rate.

1-D vectors are treated as length-N sample-based vectors, and result in
sample-based length-N row vectors.

Overlapping sections of input u are circularly convolved with the FIR
filter coefficients

$$H(z) = B(z) = b_1 + b_2 z^{-1} + \ldots + b_{n+1} z^{-n}$$

The numerator coefficients for $H(z)$ are specified as a vector
by the **FIR coefficients** parameter. The coefficient vector,
b = [b(1) b(2) ... b(n+1)], can be generated by one of the filter
design functions in the Signal Processing Toolbox, such as fir1. All
filter states are internally initialized to zero.

When either the filter coefficients or the inputs to the block are complex,
the **Output** parameter should be set to Complex. Otherwise, the default

Overlap-Save FFT Filter

Output setting, `Real`, instructs the block to take only the real part of the solution.

The circular convolution of each section is computed by multiplying the FFTs of the input section and filter coefficients, and computing the inverse FFT of the product.

```
y = ifft(fft(u(i:i+(L-1)),nfft) .* fft(b,nfft))
```

where you specify `nfft` in the **FFT size** parameter as a power of two value greater (typically *much* greater) than n+1. Values for **FFT size** that are not powers of two are rounded upwards to the nearest power-of-two value to obtain `nfft`.

The first n points of the circular convolution are invalid and are discarded. The Overlap-Save FFT Filter block outputs the remaining `nfft-n` points, which are equivalent to the linear convolution.

Latency

In *single-tasking* operation, the Overlap-Save FFT Filter block has a latency of `nfft-n+1` samples. The first `nfft-n+1` consecutive outputs from the block are zero; the first filtered input value appears at the output as sample `nfft-n+2`.

In *multitasking* operation, the Overlap-Save FFT Filter block has a latency of `2*(nfft-n+1)` samples. The first `2*(nfft-n+1)` consecutive outputs from the block are zero; the first filtered input value appears at the output as sample `2*(nfft-n)+3`.

Note For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" on page 2-56 and "Models with Multiple Sample Rates" in the Real-Time Workshop User's Guide documentation.

**Dialog
Box**

FFT size

The size of the FFT, which should be a power of two value greater than the length of the specified FIR filter.

FIR coefficients

The filter numerator coefficients.

Output

The complexity of the output; Real or Complex. When the input signal or the filter coefficients are complex, this should be set to Complex.

References Oppenheim, A. V. and R. W. Schafer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989.

Proakis, J. and D. Manolakis. *Digital Signal Processing.* 3rd ed. Englewood Cliffs, NJ: Prentice-Hall, 1996.

Overlap-Save FFT Filter

Supported Data Types

- Double-precision floating point

- Single-precision floating point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Overlap-Add FFT Filter   Signal Processing Blockset

Purpose   Overwrite submatrix or subdiagonal of input

Library   • Math Functions / Matrices and Linear Algebra / Matrix Operations

               • Signal Management / Indexing
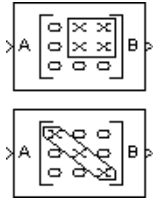
Description The Overwrite Values block overwrites a contiguous submatrix or
               subdiagonal of an input matrix. You can provide the overwriting values
               by typing them in a block parameter, or through an additional input
               port, which is useful for providing overwriting values that change at
               each time step.

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \end{bmatrix} \implies \boxed{A \quad B} \implies \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 0 & 0 & 0 \\ 1 & 2 & 0 & 0 & 0 \end{bmatrix}$$
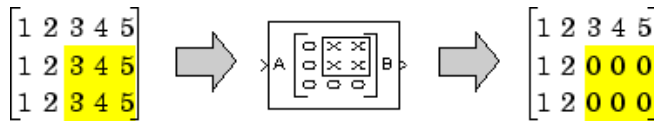
The block accepts both sample- and frame-based vectors and matrices.
The output has the same size and frame status as the original input
signal, not necessarily the same size and frame status as the signal
containing the overwriting values.

Specifying the Overwriting Values

The **Source of overwriting value(s)** parameter determines how you
must provide the overwriting values, and has the following settings.

• `Specify via dialog` – You must provide the overwriting value(s)
  in the **Overwrite with** parameter. The block uses the same
  overwriting values to overwrite the specified portion of the input at
  each time step. To learn how to specify valid overwriting values, see
  "Valid Overwriting Values" on page 10-800.

• `Second input port` – You must provide overwriting values through
  a second block input port, V. Use this setting to provide different
  overwriting values at each time step. The output inherits its size,
  rate, and frame status from the input signal, *not* the overwriting
  values.

Overwrite Values

Input signal ———▶ A

Signal of overwriting values ———▶ V

Overwrite Values

The output is the result of overwriting the **Input signal**, and has the same size, rate, and frame status as the **Input signal**.
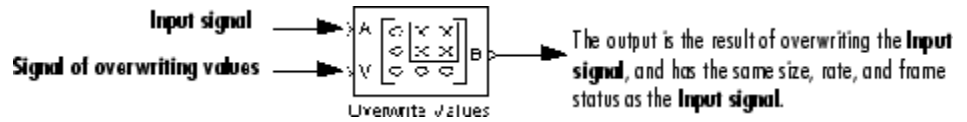
The rate at which you provide the overwriting values through input port V must match the rate at which the block receives each input matrix at input port A. The rate requirements depend on whether the input signal and overwriting values signal have the same frame status:

- When both signals are sample based, their sample rates must be the same.

- When both signals are frame based, their frame rates must be the same.

- When one signal is sample based and one signal is frame based, the sample rate of the sample-based signal must be the same as the frame rate of the frame-based signal.
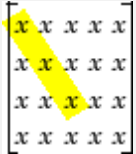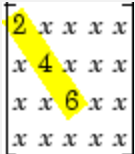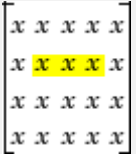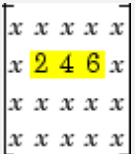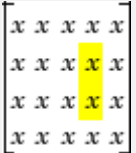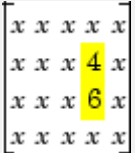
Valid Overwriting Values

The overwriting values can be a single constant, vector, or matrix, depending on the portion of the input you are overwriting, regardless of whether you provide the overwriting values through an input port or by providing them in the **Overwrite with** parameter.

Valid Overwriting Values

Portion of Input to Overwrite	Valid Overwriting Values	Example
A single element in the input	Any constant value, v	$v = 9$ $$\begin{bmatrix} x & x & x & x & x \\ x & x & x & 9 & x \\ x & x & x & x & x \\ x & x & x & x \end{bmatrix}$$

Portion of Input to Overwrite	Valid Overwriting Values	Example
A length-k portion of the diagonal $$\begin{bmatrix} x & x & x & x \\ x & x & x & x \\ x & x & x & x \\ x & x & x & x \end{bmatrix}$$	Any length-k column or row vector, v	$k = 3 \quad v = \begin{bmatrix} 2 & 4 & 6 \end{bmatrix}$ or $\begin{bmatrix} 2 \\ 4 \\ 6 \end{bmatrix}$ $$\begin{bmatrix} 2 & x & x & x \\ x & 4 & x & x \\ x & x & 6 & x \\ x & x & x & x \end{bmatrix}$$
A length-k portion of a row $$\begin{bmatrix} x & x & x & x \\ x & x & x & x \\ x & x & x & x \\ x & x & x & x \end{bmatrix}$$	Any length-k row vector, v	$k = 3 \quad v = \begin{bmatrix} 2 & 4 & 6 \end{bmatrix}$ $$\begin{bmatrix} x & x & x & x & x \\ x & 2 & 4 & 6 & x \\ x & x & x & x & x \\ x & x & x & x & x \end{bmatrix}$$
A length-k portion of a column $$\begin{bmatrix} x & x & x & x \\ x & x & x & x \\ x & x & x & x \\ x & x & x & x \end{bmatrix}$$	Any length-k column vector, v	$k = 2 \quad v = \begin{bmatrix} 4 \\ 6 \end{bmatrix}$ $$\begin{bmatrix} x & x & x & x & x \\ x & x & x & 4 & x \\ x & x & x & 6 & x \\ x & x & x & x & x \end{bmatrix}$$
An m-by-n submatrix	Any m-by-n matrix, v	

Overwrite Values

Portion of Input to Overwrite	Valid Overwriting Values	Example
$\begin{bmatrix} x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \end{bmatrix}$		$m = 2$ $n = 3$ $v = \begin{bmatrix} 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$ $\begin{bmatrix} x & x & x & x & x \\ x & x & 4 & 5 & 6 \\ x & x & 7 & 8 & 9 \\ x & x & x & x & x \end{bmatrix}$

This block supports Simulink virtual buses.
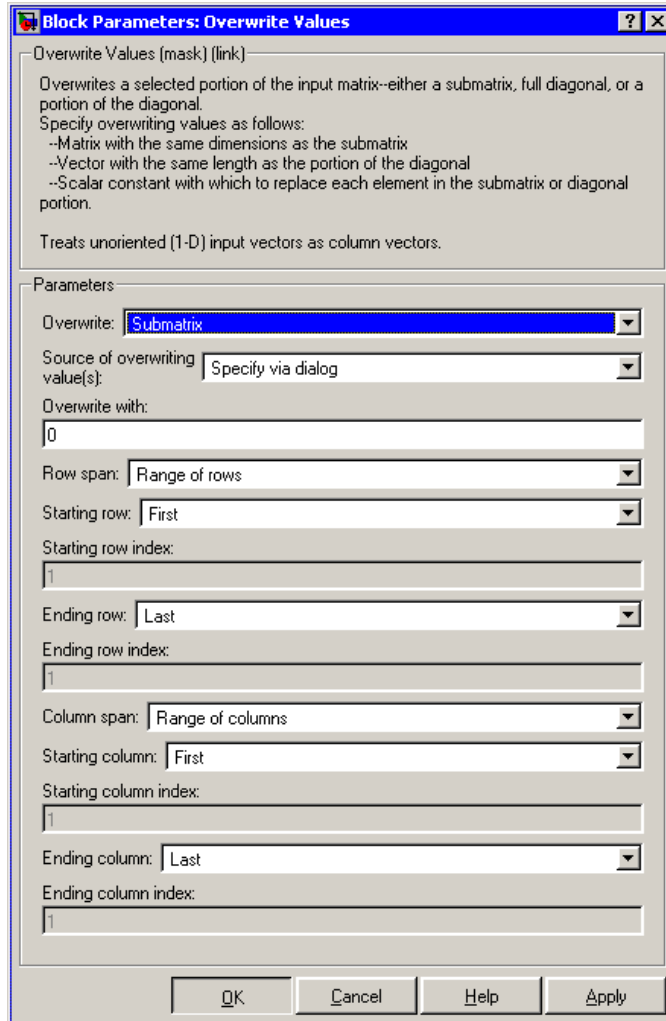
**Dialog
Box**

Note Only some of the following parameters are visible in the dialog box at any one time.

Overwrite Values

Overwrite

Determines whether to overwrite a specified submatrix or a specified portion of the diagonal.

Source of overwriting value(s)

Determines where you must provide the overwriting values: either through an input port, or by providing them in the **Overwrite with** parameter. For more information, see "Specifying the Overwriting Values" on page 10-799.

Overwrite with

The value(s) with which to overwrite the specified portion of the input matrix. Enabled only when **Source of overwriting value(s)** is set to `Specify via dialog`. To learn how to specify valid overwriting values, see "Valid Overwriting Values" on page 10-800.

Row span

The range of input rows to be overwritten. Options are `All rows`, `One row`, or `Range of rows`. For descriptions of these options, see "Dialog Box" on page 10-803.

Row/Starting row

The input row that is the first row of the submatrix that the block overwrites. For a description of the options for the **Row** and **Starting row** parameters, see Settings for Row, Column, Starting Row, and Starting Column Parameters on page 10-809. **Row** is enabled when **Row span** is set to `One row`, and **Starting row** when **Row span** is set to `Range of rows`.

Row index/Starting row index

Index of the input row that is the first row of the submatrix that the block overwrites. See how to use these parameters in Settings for Row, Column, Starting Row, and Starting Column Parameters on page 10-809. **Row index** is enabled when **Row** is set to `Index`, and **Starting row index** when **Starting row** is set to `Index`.

Row offset/Starting row offset

The offset of the input row that is the first row of the submatrix that the block overwrites. See how to use these parameters in

Settings for Row, Column, Starting Row, and Starting Column Parameters on page 10-809. **Row offset** is enabled when **Row** is set to `Offset from middle` or `Offset from last`, and **Starting row offset** is enabled when **Starting row** is set to `Offset from middle` or `Offset from last`.

Ending row

The input row that is the last row of the submatrix that the block overwrites. For a description of this parameter's options, see Settings for Ending Row and Ending Column Parameters on page 10-810. This parameter is enabled when **Row span** is set to `Range of rows`, and **Starting row** is set to any option but `Last`.

Ending row index

Index of the input row that is the last row of the submatrix that the block overwrites. See how to use this parameter in Settings for Ending Row and Ending Column Parameters on page 10-810. Enabled when **Ending row** is set to `Index`.

Ending row offset

The offset of the input row that is the last row of the submatrix that the block overwrites. See how to use this parameter in Settings for Ending Row and Ending Column Parameters on page 10-810. Enabled when **Ending row** is set to `Offset from middle` or `Offset from last`.

Column span

The range of input columns to be overwritten. Options are `All columns`, `One column`, or `Range of columns`. For descriptions of the analogous row options, see "Dialog Box" on page 10-803.

Column/Starting column

The input column that is the first column of the submatrix that the block overwrites. For a description of the options for the **Column** and **Starting column** parameters, see Settings for Row, Column, Starting Row, and Starting Column Parameters on page 10-809. **Column** is enabled when **Column span** is set to `One column`, and **Starting column** when **Column span** is set to `Range of columns`.

Overwrite Values

Column index/Starting column index

Index of the input column that is the first column of the submatrix that the block overwrites. See how to use these parameters in Settings for Row, Column, Starting Row, and Starting Column Parameters on page 10-809. **Column index** is enabled when **Column** is set to `Index`, and **Starting column index** when **Starting column** is set to `Index`.

Column offset/Starting column offset

The offset of the input column that is the first column of the submatrix that the block overwrites. See how to use these parameters in Settings for Row, Column, Starting Row, and Starting Column Parameters on page 10-809. **Column offset** is enabled when **Column** is set to `Offset from middle` or `Offset from last`, and **Starting column offset** is enabled when **Starting column** is set to `Offset from middle` or `Offset from last`.

Ending column

The input column that is the last column of the submatrix that the block overwrites. For a description of this parameter's options, see Settings for Ending Row and Ending Column Parameters on page 10-810. This parameter is enabled when **Column span** is set to `Range of columns`, and **Starting column** is set to any option but `Last`.

Ending column index

Index of the input column that is the last column of the submatrix that the block overwrites. See how to use this parameter in Settings for Ending Row and Ending Column Parameters on page 10-810. This parameter is enabled when **Ending column** is set to `Index`.

Ending column offset

The offset of the input column that is the last column of the submatrix that the block overwrites. See how to use this parameter in Settings for Ending Row and Ending Column Parameters on page 10-810. This parameter is enabled when

Ending column is set to `Offset from middle` or `Offset from last`.

Diagonal span

The range of diagonal elements to be overwritten. Options are `All elements`, `One element`, or `Range of elements`. For descriptions of these options, see "Overwriting a Subdiagonal" on page 10-813.

Element/Starting element

The input diagonal element that is the first element in the subdiagonal that the block overwrites. For a description of the options for the **Element** and **Starting element** parameters, see Element and Starting Element Parameters on page 10-813. **Element** is enabled when **Element span** is set to `One element`, and **Starting element** when **Element span** is set to `Range of elements`.

Element index/Starting element index

Index of the input diagonal element that is the first element of the subdiagonal that the block overwrites. See how to use these parameters in Element and Starting Element Parameters on page 10-813. **Element index** is enabled when **Element** is set to `Index`, and **Starting element index** when **Starting element** is set to `Index`.

Element offset/Starting element offset

The offset of the input diagonal element that is the first element of the subdiagonal that the block overwrites. See how to use these parameters in Element and Starting Element Parameters on page 10-813. **Element offset** is enabled when **Element** is set to `Offset from middle` or `Offset from last`, and **Starting element offset** is enabled when **Starting element** is set to `Offset from middle` or `Offset from last`.

Ending element

The input diagonal element that is the last element of the subdiagonal that the block overwrites. For a description of this parameter's options, see Ending Element Parameters on page 10-814. This parameter is enabled when **Element span** is set to

Overwrite Values

Range of elements, and **Starting element** is set to any option but Last.

Ending element index

Index of the input diagonal element that is the last element of the subdiagonal that the block overwrites. See how to use this parameter in Ending Element Parameters on page 10-814. This parameter is enabled when **Ending element** is set to Index.

Ending element offset

The offset of the input diagonal element that is the last element of the subdiagonal that the block overwrites. See how to use this parameter in Ending Element Parameters on page 10-814. This parameter is enabled when **Ending element** is set to Offset from middle or Offset from last.

Examples

Overwriting a Submatrix

To overwrite a submatrix, follow these steps:

1 Set the **Overwrite** parameter to Submatrix.

2 Specify the overwriting values as described in "Specifying the Overwriting Values" on page 10-799.

3 Specify which rows and columns of the input matrix are contained in the submatrix that you want to overwrite by setting the **Row span** parameter to one of the following options and the **Column span** to the analogous column-related options:

- All rows – The submatrix contains all rows of the input matrix.

- One row – The submatrix contains only one row of the input matrix, which you must specify in the **Row** parameter, as described in the following table.

- Range of rows – The submatrix contains one or more rows of the input, which you must specify in the **Starting Row** and **Ending row** parameters, as described in the following tables.

4 When you set **Row span** to One row or Range of rows, you need to further specify the row(s) contained in the submatrix by setting the **Row** or **Starting row** and **Ending row** parameters. Likewise, when you set **Column span** to One column or Range of columns, you must further specify the column(s) contained in the submatrix by setting the **Column** or **Starting column** and **Ending column** parameters. For descriptions of the settings for these parameters, see the following tables.

Settings for Row, Column, Starting Row, and Starting Column Parameters

Settings for Specifying the Submatrix's First Row or Column	First Row of Submatrix (Only row for Row span = One row)	First Column of Submatrix (Only row for Row span = One row)
First	First row of the input	First column of the input
Index	Input row specified in the **Row index** parameter	Input column specified in the **Column index** parameter
Offset from last	Input row with the index M - rowOffset where M is the number of input rows, and rowOffset is the value of the **Row offset** or **Starting row offset** parameter	Input column with the index N - colOffset where N is the number of input columns, and colOffset is the value of the **Column offset** or **Starting column offset** parameter
Last	Last row of the input	Last column of the input

Overwrite Values

Settings for Specifying the Submatrix's First Row or Column	First Row of Submatrix (Only row for Row span = One row)	First Column of Submatrix (Only row for Row span = One row)
`Offset from middle`	Input row with the index `floor(M/2 + 1 - rowOffset)` where `M` is the number of input rows, and `rowOffset` is the value of the **Row offset** or **Starting row offset** parameter	Input column with the index `floor(N/2 + 1 - rowOffset)` where `N` is the number of input columns, and `colOffset` is the value of the or **Column offset** or **Starting column offset** parameter
`Middle`	Input row with the index `floor(M/2 + 1)` where `M` is the number of input rows	Input columns with the index `floor(N/2 + 1)` where `N` is the number of input columns

Settings for Ending Row and Ending Column Parameters

Settings for Specifying the Submatrix's Last Row or Column	Last Row of Submatrix	Last Column of Submatrix
`Index`	Input row specified in the **Ending row index** parameter	Input column specified in the **Ending column index** parameter
`Offset from last`	Input row with the index `M - rowOffset` where `M` is the number of input rows, and `rowOffset` is the value of the **Ending row offset** parameter	Input column with the index `N - colOffset` where `N` is the number of input columns, and `colOffset` is the value of the **Ending column offset** parameter

Settings for Specifying the Submatrix's Last Row or Column	Last Row of Submatrix	Last Column of Submatrix
Last	Last row of the input	Last column of the input
Offset from middle	Input row with the index `floor(M/2 + 1 - rowOffset)` where M is the number of input rows, and `rowOffset` is the value of the **Ending row offset** parameter	Input column with the index `floor(N/2 + 1 - rowOffset)` where N is the number of input columns, and `colOffset` is the value of the **Ending column offset** parameter
Middle	Input row with the index `floor(M/2 + 1)` where M is the number of input rows	Input columns with the index`floor(N/2 + 1)`where N is the number of input columns

For example, to overwrite the lower-right 2-by-3 submatrix of a 3-by-5 input matrix with all zeros, enter the following set of parameters:

- **Overwrite** = Submatrix
- **Source of overwriting value(s)** = Specify via dialog
- **Overwrite with** = 0
- **Row span** = Range of rows
- **Starting row** = Index
- **Starting row index** = 2
- **Ending row** = Last
- **Column span** = Range of columns
- **Starting column** = Offset from last
- **Starting column offset** = 2

Overwrite Values

- **Ending column** = Last

  The following figure shows the block with the above settings overwriting a portion of a 3-by-5 input matrix.
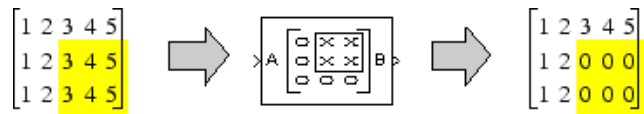
  $$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \end{bmatrix} \Rightarrow \boxed{A \; \boxed{} \; B} \Rightarrow \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 0 & 0 & 0 \\ 1 & 2 & 0 & 0 & 0 \end{bmatrix}$$

  There are often several possible parameter combinations that select the *same* submatrix from the input. For example, instead of specifying Last for **Ending column**, you could select the same submatrix by specifying

  - **Ending column** = Index
  - **Ending column index** = 5

Overwriting a Subdiagonal

To overwrite a subdiagonal, follow these steps:

1 Set the **Overwrite** parameter to Diagonal.

2 Specify the overwriting values as described in "Specifying the Overwriting Values" on page 10-799.

3 Specify the subdiagonal that you want to overwrite by setting the **Diagonal span** parameter to one of the following options:

- All elements–Overwrite the entire input diagonal.

- One element–Overwrite one element in the diagonal, which you must specify in the **Element** parameter (described below).

- Range of elements – Overwrite a portion of the input diagonal, which you must specify in the **Starting element** and **Ending element** parameters, as described in the following table.

4 When you set **Diagonal span** to One element or Range of elements, you need to further specify which diagonal element(s) to overwrite by setting the **Element** or **Starting element** and **Ending element** parameters. See the following tables.

Element and Starting Element Parameters

Settings for Element and Starting Element Parameters	First Element in Subdiagonal (Only element when Diagonal span = One element)
First	Diagonal element in first row of the input
Index	kth diagonal element, where k is the value of the **Element index** or **Starting element index** parameter

Overwrite Values

Settings for Element and Starting Element Parameters	First Element in Subdiagonal (Only element when Diagonal span = One element)
`Offset from last`	Diagonal element in the row with the index `M - offset` where `M` is the number of input rows, and `offset` is the value of the **Element offset** or **Starting element offset** parameter
`Last`	Diagonal element in the last row of the input
`Offset from middle`	Diagonal element in the input row with the index `floor(M/2 + 1 - offset)` where `M` is the number of input rows, and `offset` is the value of the **Element offset** or **Starting element offset** parameter
`Middle`	Diagonal element in the input row with the index `floor(M/2 + 1)` where `M` is the number of input rows

Ending Element Parameters

Settings for Ending Element Parameter	Last Element in Subdiagonal
`Index`	kth diagonal element, where k is the value of the **Ending element index** parameter
`Offset from last`	Diagonal element in the row with the index `M - offset` where `M` is the number of input rows, and `offset` is the value of the **Ending element offset** parameter
`Last`	Diagonal element in the last row of the input

Settings for Ending Element Parameter	Last Element in Subdiagonal
Offset from middle	Diagonal element in the input row with the index `floor(M/2 + 1 - offset)` where M is the number of input rows, and `offset` is the value of the **Ending element offset** parameter
Middle	Diagonal element in the input row with the index `floor(M/2 + 1)` where M is the number of input rows

Supported Data Types

Port	Supported Data Types
A	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Custom data types • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
V	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Custom data types • Boolean • 8-, 16-, and 32-bit signed integers

Overwrite Values

Port	Supported Data Types
	• 8-, 16-, and 32-bit unsigned integers
B	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Custom data types
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

To learn how to convert your data types to the above data types in
MATLAB and Simulink, see "Supported Data Types and How to Convert
to Them" on page 7-2.

See Also

Reshape	Simulink
Selector	Simulink
Submatrix	Signal Processing Blockset
Variable Selector	Signal Processing Blockset
reshape	MATLAB

Purpose   Alter input dimensions by padding or truncating rows and/or columns

Library   Signal Operations

Description The Pad block changes the dimensions of the input matrix from
M_i-by-N_i to M_o-by-N_o by padding or truncating along the columns, rows, or columns and rows. Use the **Pad along** parameter to specify the dimensions to change.

Use the **Value** parameter to specify the value with which to pad your input matrix.

Using the **Pad signal at** parameter, you can choose to pad your input matrix at the end or the beginning of a row and/or column.

The **Number of output rows** and/or **Number of output columns** parameters refer to the dimensions of the output, M_o and N_o. You can set these parameters to User-specified or Next power of two. When you choose User-specified, enter a scalar value in the **Specified number of output rows** and/or **Specified number of output columns** parameters. When you choose Next power of two, the block pads the input matrix along the columns and/or rows until the length of the columns and/or rows is equal to a power of two. When the length of the input matrix's columns and/or rows is already equal to a power of two, the block does not pad the input matrix.

When you choose User-specified for the **Number of output rows** and/or **Number of output columns** parameters, you can specify a scalar value in the **Specified number of output rows** and/or **Specified number of output columns** parameters that truncates the size of your input matrix. The following options are available for the **Action when truncation occurs** parameter:

• None – Select this option when you do not want to be notified that the input matrix is truncated.

• Warning – Choose this option when you want a warning to be displayed in the MATLAB Command Window when the input matrix is truncated.

- Error – Click this option when you want an error dialog box to be displayed and the simulation terminated when the input matrix is truncated.

The behavior of the Pad block and Zero Pad block is identical, with the exception that the Pad block can pad the input matrix with values other than zero. See the Zero Pad block reference page for more information on the behavior of the Zero Pad block.

Dialog Box
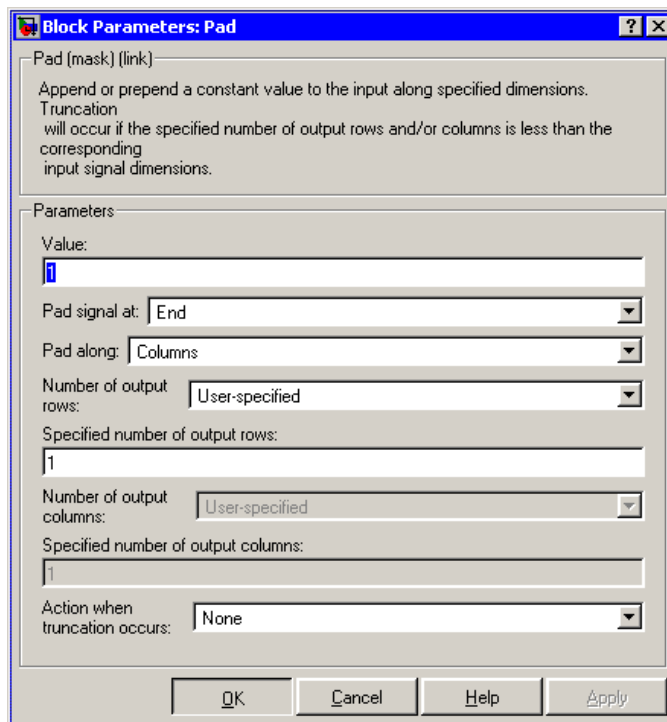
Value

The scalar value with which to pad the input matrix. Tunable.

Pad signal at

    The input matrix can be padded at the beginning of the rows and/or columns or at the end of the rows and/or columns.

Pad along

    The direction along which to pad or truncate. `Columns` specifies that the *row* dimension should be changed to M_o. `Rows` specifies that the *column* dimension should be changed to N_o. `Columns and rows` specifies that both column and row dimensions should be changed. `None` disables padding and truncation and passes the input through to the output unchanged.

Number of output rows

    The total number of output rows. When you select `User-specified`, type a scalar value in the **Specified Number of output rows** parameter. When you select `Next power of two`, the block pads the columns of the input matrix until the number of rows is equal to a power of two. When the number of rows is already equal to a power of two, the block does not pad the input matrix.

Specified number of output rows

    The desired number of rows in the output, M_o. This parameter is enabled when you select `Columns` or `Columns and rows` in the **Pad along** menu and `User-specified` is chosen in the **Number of output rows** parameter.

Number of output columns

    The total number of output columns. When you select `User-specified`, type a scalar value in the **Specified Number of output columns** parameter. When you select `Next power of two`, the block pads the rows of the input matrix until the number of columns is equal to a power of two. When the number of columns is already equal to a power of two, the block does not pad the input matrix.

Specified number of output columns

    The desired number of columns in the output, N_o. This parameter is enabled when you select `Rows` or `Columns and rows` in the **Pad**

Pad

along menu and User-specified is chosen in the **Number of output columns** parameter.

Action when truncation occurs

Choose None when you do not want to be notified that the input matrix is truncated. Select Warning to display a warning when the input matrix is truncated. Choose Error when you want an error dialog box to be displayed and the simulation terminated when the input matrix is truncated.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating-point
	• Single-precision floating-point
	• Fixed point (signed and unsigned)
	• Custom data types
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating-point
	• Single-precision floating-point
	• Fixed point (signed and unsigned)
	• Custom data types
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Matrix Concatenation	Simulink
Repeat	Signal Processing Blockset
Submatrix	Signal Processing Blockset
Upsample	Signal Processing Blockset
Variable Selector	Signal Processing Blockset
Zero Pad	Signal Processing Blockset

Peak Finder

Purpose   Determine whether each value of input signal is local minimum or maximum

**Library**   Signal Operations

Description
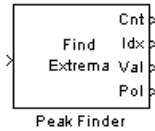
The Peak Finder block outputs the number of local extrema in the input signal at the Cnt port. Optionally, it can also output the extrema indices, the extrema values, and a binary indicator of whether or not the extrema are maxima or minima.

The Peak Finder block compares the current signal value to the previous and next values to determine if the current value is an extremum. Use the **Peak type(s)** parameter to specify whether you are looking for maxima, minima, or both.

If you select the **Output peak indices** check box, the Idx port appears on the block. The block outputs the zero-based extrema indices at the Idx port. If you select the **Output peak values** check box, the Val port appears on the block. The block outputs the extrema values at the Val port. If you select either of these check boxes and `Maxima and Minima` is selected for the **Peak type(s)**, the Pol port also appears on the block. If the signal value is a maximum, the block outputs a 1 at the Pol ("Polarity") port. If the signal value is a minimum, the block outputs a 0 at the Pol port.

Note that nothing is output at the Idx, Val, and Pol ports for an input signal value that is not an extremum.

Use the **Maximum number of peaks to find** parameter to specify how many extrema to look for in each input signal. The block stops searching the input signal once this maximum number of extrema has been found.

If you select the **Ignore peaks within threshold of neighboring values** check box, the block no longer detects low-amplitude peaks. This feature allows the block to ignore noise within a threshold value that you define. Enter a threshold value for the **Threshold** parameter. Now, the current value is a maximum if $(\text{current} - \text{previous}) > \text{threshold}$ and $(\text{current} - \text{next}) > \text{threshold}$. The current value is a minimum if $(\text{current} - \text{previous}) < -\text{threshold}$ and $(\text{current} - \text{next}) < -\text{threshold}$.

This block supports single-channel, frame-based, and sample-based inputs. These input signals must be real-valued fixed-point or floating-point scalars or vectors.

Examples

Example 1

Consider the input vector

```
[9 6 10 3 4 5 0 12]
```

The table below shows the analysis made by the Peak Finder block. Note that the first and last input signal values are not considered:

Previous, current, and next values	9 6 10	6 10 3	10 3 4	3 4 5	4 5 0	5 0 12
Current value if it is an extremum	6	10	3	–	5	0
Index of current value if it is an extremum	1	2	3	–	5	6
Polarity of current value if it is an extremum	0	1	0	–	1	0

Therefore, for this example the outputs at the block ports are

Cnt: 5

Idx: [1 2 3 5 6]

Val: [6 10 3 5 0]

Pol: [0 1 0 1 0]

Example 2

Note that the **Overflow mode** parameter can affect the output of the block when the input is fixed point. Consider the following model:

Peak Finder
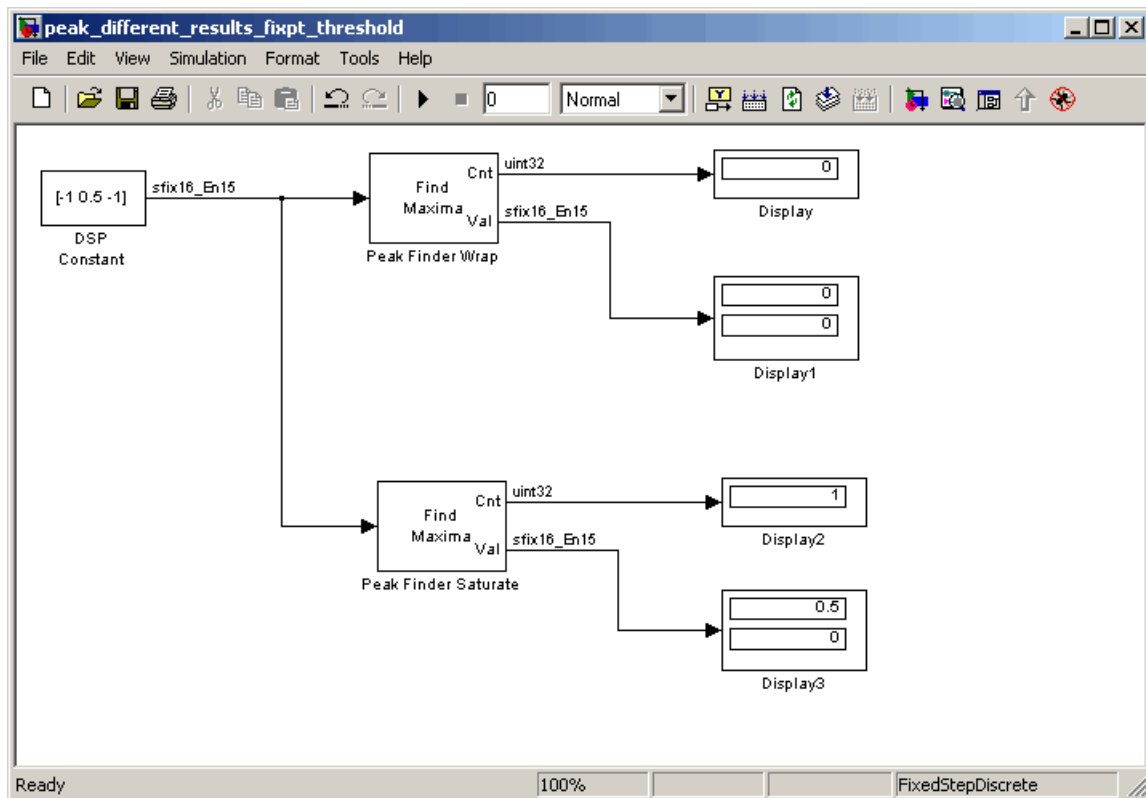
In this model, the settings in the DSP Constant block are

- **Constant value** – [-1 0.5 -1]
- **Sample mode** – Discrete
- **Output** – Sample-based
- **Sample time** – 1
- **Output data type** – Fixed-point
- **Signed** – selected

- **Word length** – `16`
- **Set fraction length in output to** – `User-defined`
- **Fraction length** – `15`

The settings in the Peak Finder blocks are

- **Peak type(s)** – `Maxima`
- **Output peak indices** – not selected
- **Output peak values** – selected
- **Maximum number of peaks to find** – `2`
- **Ignore peaks within threshold of neighboring values** – selected
- **Threshold** – `0.25`
- **Rounding mode** – `Floor`
- **Overflow mode** – `Wrap` for Peak Finder Wrap, `Saturate` for Peak Finder Saturate

Setting the **Overflow mode** parameter of the Peak Finder Wrap block to `Wrap` causes the calculations $(current - previous) > threshold$ and $(current - next) > threshold$ to wrap on overflow, thereby causing the maximum to be missed.

Peak Finder

Dialog Box

The **Main** pane of the Peak Finder block dialog appears as follows:
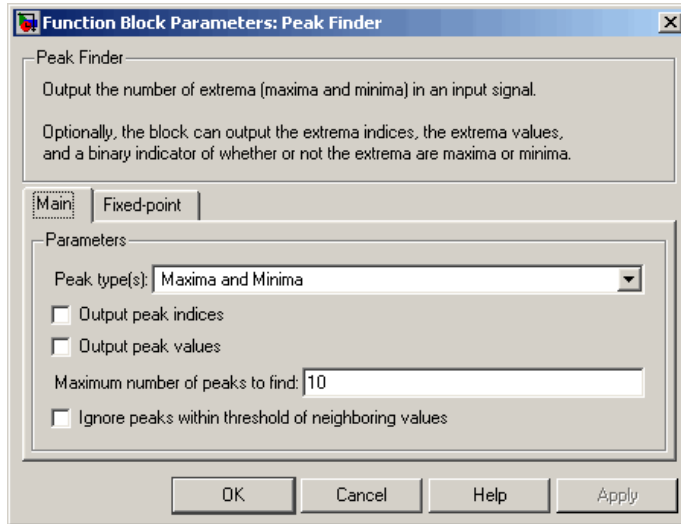
Peak type(s)
> Specify whether you are looking for maxima, minima, or both.

Output peak indices
> Select this check box if you want the block to output the extrema indices at the Idx port.

Output peak values
> Select this check box if you want the block to output the extrema values at the Val port.

Maximum number of peaks to find
> Enter the number of extrema to look for in each input signal. The block stops searching the input signal for extrema once the maximum number of extrema has been found. The value of this parameter must be an integer greater than or equal to one.

Ignore peaks within threshold of neighboring values
Select this check box if you want to eliminate the detection of peaks whose amplitudes are within a specified threshold of neighboring values.

Threshold
Enter your threshold value. This parameter appears if you select the **Ignore peaks within threshold of neighboring values** check box.

The **Fixed-point** pane of the Peak Finder block dialog appears as follows:
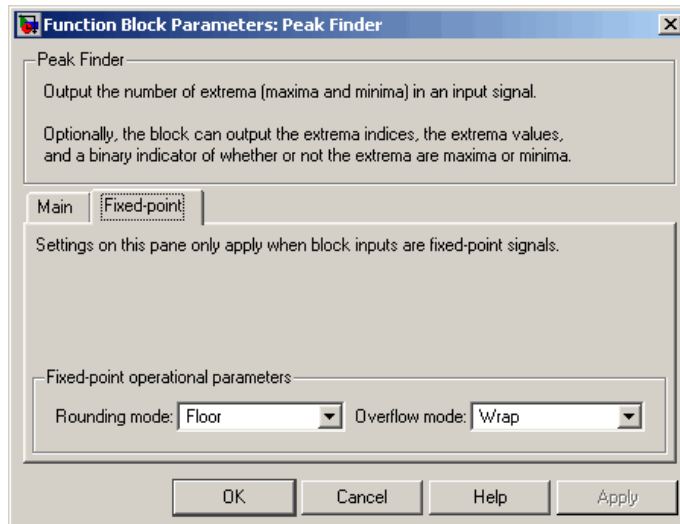
Rounding mode
Select the rounding mode to be used when block inputs are fixed point.

Overflow mode
Select the overflow mode to be used when block inputs are fixed point.

Peak Finder

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Cnt	• 32-bit unsigned integers
Idx	• 32-bit unsigned integers
Val	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Pol	• Boolean

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Maximum	Signal Processing Blockset
Minimum	Signal Processing Blockset

Purpose   Compute nonparametric estimate of the spectrum

Library   Estimation / Power Spectrum Estimation
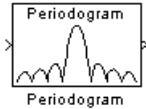
Description The Periodogram block computes a nonparametric estimate of the spectrum. The block averages the squared magnitude of the FFT computed over windowed sections of the input and normalizes the spectral average by the square of the sum of the window samples.

Both an M-by-N frame-based matrix input and an M-by-N sample-based matrix input are treated as M sequential time samples from N independent channels. The block computes a separate estimate for each of the N independent channels and generates an N_{fft}-by-N matrix output. When you select the **Inherit FFT length from input dimensions** check box, N_{fft} is specified by the frame size of the input, which must be a power of 2. When you clear the **Inherit FFT length from input dimensions** check box, N_{fft} is specified as a power of 2 by the **FFT length** parameter, and the block zero pads or truncates the input to N_{fft} before computing the FFT.

Each column of the output matrix contains the estimate of the corresponding input column's power spectral density at N_{fft} equally spaced frequency points in the range $[0,F_s)$, where F_s is the signal's sample frequency. The output is always sample based.

The **Number of spectral averages** specifies the number of spectra to average. Setting this parameter to 1 effectively disables averaging.

The **Window type**, **Stopband ripple**, **Beta**, and **Window sampling** parameters all apply to the specification of the window function; see the Window Function block reference page for more details on these four parameters.

Example   The dspstfft demo provides an illustration of using the Periodogram and Matrix Viewer blocks to create a spectrogram. The dspsacomp demo compares the Periodogram block with several other spectral estimation methods.

Periodogram

**Dialog
Box**

Window type
> Enter the type of window to apply. See the Window Function block reference page for more details. Tunable.

Stopband attenuation in dB
> Enter the level, in dB, of stopband attenuation, R_s, for the Chebyshev window. This parameter is enabled if, for the **Window type** parameter, you choose Chebyshev. Tunable.

Beta
> Enter the β parameter for the Kaiser window. This parameter is enabled if, for the **Window type** parameter, you chose Kaiser. Increasing **Beta** widens the mainlobe and decreases the amplitude of the window sidelobes in the window's frequency magnitude response. Tunable.

Window sampling

From the list, choose Symmetric or Periodic. Tunable.

Inherit FFT length from input dimensions

When you select this check box, the block uses the input frame size as the number of data points, N_{fft}, on which to perform the FFT.

FFT length

Enter the number of data points, N_{fft}, on which to perform the FFT. When N_{fft} exceeds the input frame size, the frame is zero-padded as needed. This parameter is enabled when you clear the **Inherit FFT length from input dimensions** check box.

Number of spectral averages

Enter the number of spectra to average; setting this parameter to 1 disables averaging.

References   Oppenheim, A. V. and R. W. Schafer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989.

Proakis, J. and D. Manolakis. *Digital Signal Processing.* 3rd ed. Englewood Cliffs, NJ: Prentice-Hall, 1996.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
Output	• Double-precision floating point
	• Single-precision floating point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

Periodogram

See Also

Burg Method	Signal Processing Blockset
Inverse Short-Time FFT	Signal Processing Blockset
Magnitude FFT	Signal Processing Blockset
Short-Time FFT	Signal Processing Blockset
Spectrum Scope	Signal Processing Blockset
Window Function	Signal Processing Blockset
Yule-Walker Method	Signal Processing Blockset
pwelch	Signal Processing Toolbox

See "Power Spectrum Estimation" on page 6-6 for related information.

Purpose   Reorder matrix rows or columns

Library   Math Functions / Matrices and Linear Algebra / Matrix Operations
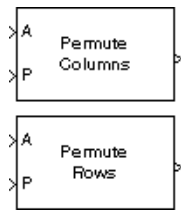
Description   The Permute Matrix block reorders the rows or columns of M-by-N input matrix A as specified by indexing input P.

When the **Permute** parameter is set to Rows, the block uses the rows of A to create a new matrix with the same column dimension. Input P is a length-L vector whose elements determine where each row from A should be placed in the L-by-N output matrix.

```
% Equivalent MATLAB code
y = [A(P(1),:) ; A(P(2),:) ; A(P(3),:) ; ... ; A(P(end),:)]
```

For row permutation, a length-M 1-D vector input at the A port is treated as a M-by-1 matrix.

When the **Permute** parameter is set to Columns, the block uses the columns of A to create a new matrix with the same row dimension. Input P is a length-L vector whose elements determine where each column from A should be placed in the M-by-L output matrix.

```
% Equivalent MATLAB code
y = [A(:,P(1)) A(:,P(2)) A(:,P(3)) ... A(:,P(end))]
```

For column permutation, a length-N 1-D vector input at the A port is treated as a 1-by-N matrix.

When an index value in input P references a nonexistent row or column of matrix A, the block reacts with the behavior specified by the **Invalid permutation index** parameter. The following options are available:

- Clip index – Clip the index to the nearest valid value (1 or M for row permutation, and 1 or N for column permutation), and *do not* issue an alert. Example: For a 3-by-7 input matrix, a column index of 9 is clipped to 7, and a row index of -2 is clipped to 1.

Permute Matrix

- `Clip and warn` – Display a warning message in the MATLAB command window, and clip the index as described above.

- `Generate error` – Display an error dialog box and terminate the simulation.

When length of the permutation vector `P` is not equal to the number of rows or columns of the input matrix `A`, you can choose to get an error dialog box and terminate the simulation by selecting **Error when length of P is not equal to Permute dimension size**.

When input `A` is frame based, the output is frame based; otherwise, the output is sample based.

Examples   In the model below, the top Permute Matrix block places the second row of the input matrix in both the first and fifth rows of the output matrix, and places the third row of the input matrix in the three middle rows of the output matrix. The bottom Permute Matrix block places the second column of the input matrix in both the first and fifth columns of the output matrix, and places the third column of the input matrix in the three middle columns of the output matrix.
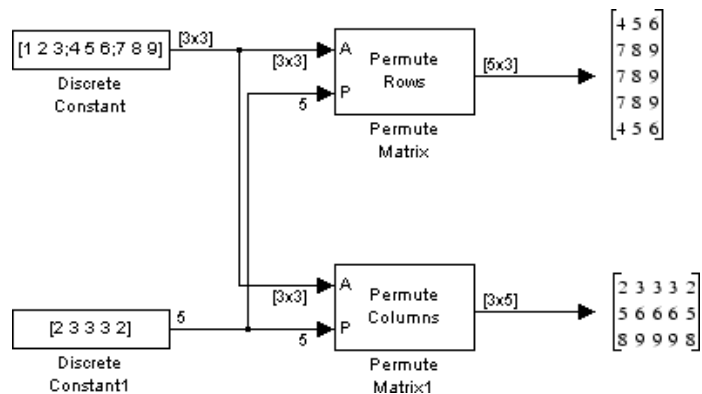
As shown in the example above, rows and columns of `A` can appear any number of times in the output, or not at all.

**Dialog
Box**
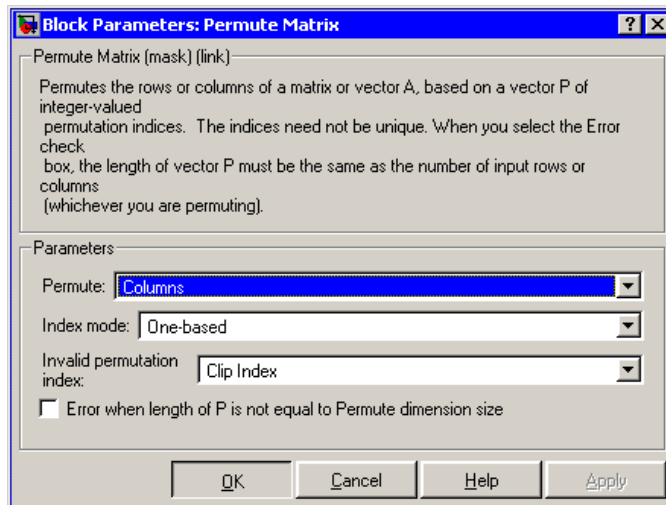
Permute

Method of constructing the output matrix; by permuting rows or columns of the input.

Index mode

When set to One-based, a value of 1 in the permutation vector P refers to the first row or column of the input matrix A. When set to Zero-based, a value of 0 in P refers to the first row or column of A.

Invalid permutation index

Response to an invalid index value. Tunable.

Error when length of P is not equal to Permute dimension size

Option to display an error dialog box and terminate the simulation when the length of the permutation vector P is not equal to the number of rows or columns of the input matrix A.

Permute Matrix

Supported Data Types

Port	Supported Data Types
A	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Custom data types
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
P	• Double-precision floating point
	• Single-precision floating point
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Custom data types
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

To learn how to convert your data types to the above data types in
MATLAB and Simulink, see "Supported Data Types and How to Convert
to Them" on page 7-2.

See Also

Submatrix	Signal Processing Blockset
Transpose	Signal Processing Blockset
Variable Selector	Signal Processing Blockset
permute	MATLAB

See "Reordering Channels in Multichannel Frame-Based Signals" on page 1-51 for related information.

Polynomial Evaluation

Purpose   Evaluate polynomial expression

Library   Math Functions / Polynomial Functions

Description

The Polynomial Evaluation block applies a polynomial function to the real or complex input at the In port.
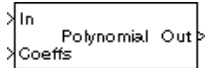
```
y = polyval(u)      % Equivalent MATLAB code
```

The Polynomial Evaluation block performs these types of operation more efficiently than the equivalent construction using Simulink Sum and Math Function blocks.

When you select the **Use constant coefficients** check box, you specify the polynomial expression in the **Constant coefficients** parameter. When you do not select **Use constant coefficients**, a variable polynomial expression is specified by the input to the Coeffs port. In both cases, the polynomial is specified as a vector of real or complex coefficients in order of descending exponents.

The table below shows some examples of the block's operation for various coefficient vectors.

Coefficient Vector	Equivalent Polynomial Expression
`[1 2 3 4 5]`	$y = u^4 + 2u^3 + 3u^2 + 4u + 5$
`[1 0 3 0 5]`	$y = u^4 + 3u^2 + 5$
`[1 2+i 3 4-3i 5i]`	$y = u^4 + (2+i)u^3 + 3u^2 + (4-3i)u + 5i$

Each element of a vector or matrix input to the In port is processed independently, and the output size and frame status are the same as the input.

**Dialog
Box**

Block Parameters: Polynomial Evaluation

Polynomial Evaluation (mask)

Apply specified polynomial function to input.

Example: [1 3 5] represents u^2 + 3u + 5.

Parameters

☑ Use constant coefficients

Constant coefficients:

[5 4 3 2 1]

| OK | Cancel | Help | Apply |
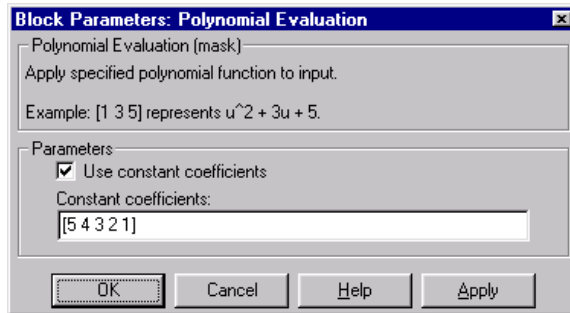
Use constant coefficients

When selected, enables the **Constant coefficients** parameter
and disables the Coeffs input port.

Constant coefficients

The vector of polynomial coefficients to apply to the input, in order
of descending exponents. This parameter is enabled when you
select the **Use constant coefficients** check box.

**Supported
Data
Types**

- Double-precision floating point

- Single-precision floating point

To learn how to convert your data types to the above data types in
MATLAB and Simulink, see "Supported Data Types and How to Convert
to Them" on page 7-2.

See Also

Least Squares Polynomial Fit	Signal Processing Blockset
Math Function	Simulink
Sum	Simulink
polyval	MATLAB

Polynomial Stability Test

Purpose

Use Schur-Cohn algorithm to determine whether all roots of input polynomial are inside unit circle

Library

Math Functions / Polynomial Functions

Description

The Polynomial Stability Test block uses the Schur-Cohn algorithm to determine whether all roots of a polynomial are within the unit circle.
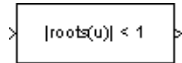
> |roots(u)| < 1

```
y = all(abs(roots(u)) < 1)          % Equivalent MATLAB code
```

Each column of the M-by-N input matrix u contains M coefficients from a distinct polynomial,

$$f(x) = u_1 x^{M-1} + u_2 x^{M-2} + \cdots + u_M$$

arranged in order of descending exponents, $u_1, u_2, ..., u_M$. The polynomial has order M-1 and positive integer exponents.

Inputs can be frame based or sample based, and both represent the polynomial coefficients as shown above. For convenience, a length-M 1-D vector input is treated as an M-by-1 matrix.

The output is a 1-by-N matrix with each column containing the value 1 or 0. The value 1 indicates that the polynomial in the corresponding column of the input is stable; that is, the magnitudes of all solutions to $f(x) = 0$ are less than 1. The value 0 indicates that the polynomial in the corresponding column of the input might be unstable; that is, the magnitude of at least one solution to $f(x) = 0$ is greater than or equal to 1.

The output is always sample based.

Applications

This block is most commonly used to check the pole locations of the denominator polynomial, $A(z)$, of a transfer function, $H(z)$.

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_1 + b_2 z^{-1} + \ldots + b_m z^{-(m-1)}}{a_1 + a_2 z^{-1} + \ldots + a_n z^{-(n-1)}}$$

The poles are the n-1 roots of the denominator polynomial, $A(z)$. When any poles are located outside the unit circle, the transfer function $H(z)$ is unstable. As is typical in DSP applications, the transfer function above is specified in descending powers of z^{-1} rather than z.

Dialog Box	**Block Parameters: Polynomial Stability Test** ☒ ─ Polynomial Stability Test (mask) ────────── Determines if all roots of the input polynomial lie inside the unit circle. Implemented using the Schur-Cohn algorithm. [OK] [Cancel] [Help] [Apply]
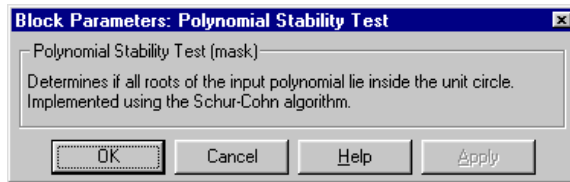
Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Boolean – Block outputs are always Boolean.

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Least Squares Polynomial Fit	Signal Processing Blockset
Polynomial Evaluation	Signal Processing Blockset
polyfit	MATLAB

Pseudoinverse

Purpose   Compute Moore-Penrose pseudoinverse of matrix

Library   Math Functions / Matrices and Linear Algebra / Matrix Inverses

Description   The Pseudoinverse block computes the Moore-Penrose pseudoinverse of input matrix A.
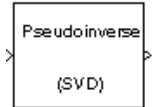
```
[U,S,V] = svd(A,0)        % Equivalent MATLAB code
```

The pseudoinverse of A is the matrix A^+ such that

$$A^+ = VS^+U^*$$

where U and V are orthogonal matrices, and S is a diagonal matrix. The pseudoinverse has the following properties:

- $AA^+ = (AA^+)^*$

- $A^+A = (A^+A)^*$

- $AA^+A = A$

- $A^+AA^+ = A^+$

The output is always sample based.

Dialog Box

Block Parameters: Pseudoinverse

Pseudoinverse (mask) (link)

X is the Moore-Penrose Pseudoinverse and satisfies 4 criteria:
X*A*X=X, A*X*A=A, and A*X and X*A are Hermitian.

OK   Cancel   Help   Apply

References   Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

Supported Data Types

- Double-precision floating point

- Single-precision floating point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Cholesky Inverse	Signal Processing Blockset
LDL Inverse	Signal Processing Blockset
LU Inverse	Signal Processing Blockset
Singular Value Decomposition	Signal Processing Blockset
inv	MATLAB

See "Inverting Matrices" on page 6-10 for related information.

QR Factorization

Purpose

Factor rectangular matrix into unitary and upper triangular components

Library

Math Functions / Matrices and Linear Algebra / Matrix Factorizations

Description
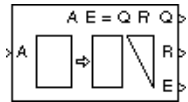
The QR Factorization block uses a modified Gram-Schmidt iteration to factor a column permutation of the M-by-N input matrix A as

$$A_e = QR$$

where Q is an M-by-min(M,N) unitary matrix, and R is a min(M,N)-by-N upper-triangular matrix. A length-M vector input is treated as an M-by-1 matrix, and is always sample based.

The column-pivoted matrix A_e contains the columns of A permuted as indicated by the contents of length-N permutation vector E.

```
Ae = A(:,E)     % Equivalent MATLAB code
```

The block selects a column permutation vector E, which ensures that the diagonal elements of matrix R are arranged in order of decreasing magnitude.

$$|r_{i+1,j+1}| > |r_{i,j}| \qquad i = j$$

QR factorization is an important tool for solving linear systems of equations because of good error propagation properties and the invertability of unitary matrices.

$$Q^{-1} = Q^*$$

Unlike LU and Cholesky factorizations, the matrix A does not need to be square for QR factorization. Note, however, that QR factorization requires twice as many operations as Gaussian elimination.

Examples

A sample factorization is shown below. The input to the block is matrix A, which is permuted according to vector E to produce matrix A_e. Matrix A_e is factored to produce the Q and R output matrices.
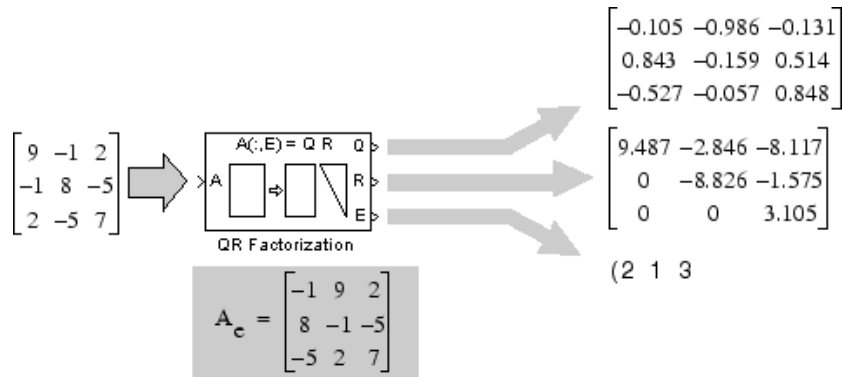
Dialog Box
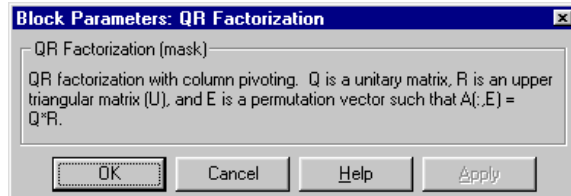
References   Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

Supported Data Types

- Double-precision floating point
- Single-precision floating point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Cholesky Factorization	Signal Processing Blockset
LU Factorization	Signal Processing Blockset

QR Factorization

QR Solver	Signal Processing Blockset
Singular Value Decomposition	Signal Processing Blockset
qr	MATLAB

See "Factoring Matrices" on page 6-9 for related information.

Purpose Find minimum-norm-residual solution to A*X*=B

Library Math Functions / Matrices and Linear Algebra / Linear System Solvers
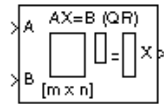
Description The QR Solver block solves the linear system A*X*=B, which can be overdetermined, underdetermined, or exactly determined. The system is solved by applying QR factorization to the M-by-N matrix, A, at the A port. The input to the B port is the right side M-by-L matrix, B. A length-M 1-D vector input at either port is treated as an M-by-1 matrix.

The output at the x port is the N-by-L matrix, X. X is always sample based, and is chosen to minimize the sum of the squares of the elements of B-AX. When B is a vector, this solution minimizes the vector 2-norm of the residual (B-AX is the residual). When B is a matrix, this solution minimizes the matrix Frobenius norm of the residual. In this case, the columns of X are the solutions to the L corresponding systems $AX_k=B_k$, where B_k is the kth column of B, and X_k is the kth column of *X*.

X is known as the minimum-norm-residual solution to AX=B. The minimum-norm-residual solution is unique for overdetermined and exactly determined linear systems, but it is not unique for underdetermined linear systems. Thus when the QR Solver is applied to an underdetermined system, the output *X* is chosen such that the number of nonzero entries in X is minimized.

Algorithm QR factorization factors a column-permuted variant (A_e) of the M-by-N input matrix A as

$$A_e = QR$$

where Q is a M-by-min(M,N) unitary matrix, and R is a min(M,N)-by-N upper-triangular matrix.

The factored matrix is substituted for A_e in

$$A_e X = B_e$$

and

QR Solver

$$QRX = B_e$$

is solved for X by noting that $Q^{-1} = Q^*$ and substituting $Y = Q^* B_e$. This requires computing a matrix multiplication for Y and solving a triangular system for X.

$$RX = Y$$

Dialog Box
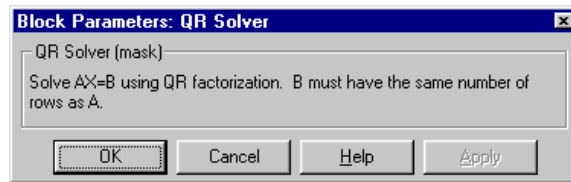
Block Parameters: QR Solver

QR Solver (mask)

Solve AX=B using QR factorization. B must have the same number of rows as A.

OK   Cancel   Help   Apply

Supported Data Types

- Double-precision floating point
- Single-precision floating point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Levinson-Durbin	Signal Processing Blockset
LDL Solver	Signal Processing Blockset
LU Solver	Signal Processing Blockset
QR Factorization	Signal Processing Blockset
SVD Solver	Signal Processing Blockset

See "Solving Linear Systems" on page 6-7 for related information.

Purpose   Store inputs in FIFO register

Library   Signal Management / Buffers
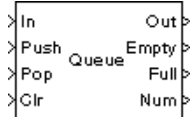
Description   The Queue block stores a sequence of input samples in a first in, first out (FIFO) register. The register capacity is set by the **Register size** parameter, and inputs can be scalars, vectors, or matrices.

The block *pushes* the input at the In port onto the end of the queue when a trigger event is received at the Push port. When a trigger event is received at the Pop port, the block *pops* the first element off the queue and holds the Out port at that value. The first input to be pushed onto the queue is always the first to be popped off.
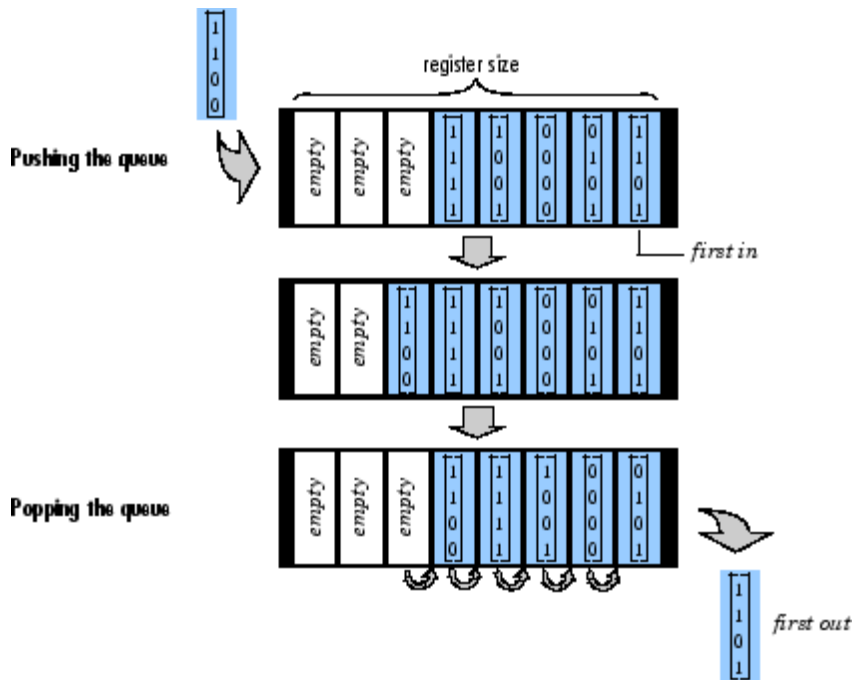
Queue

A trigger event at the optional Clr port (enabled by the **Clear input** check box) empties the queue contents. When you select **Clear output port on reset**, then a trigger event at the Clr port empties the queue *and* sets the value at the Out port to zero. This setting also applies when a disabled subsystem containing the Queue block is reenabled; the Out port value is only reset to zero in this case when you select **Clear output port on reset**.

When two or more of the control input ports are triggered at the same time step, the operations are executed in the following order:

1 Clr

2 Push

3 Pop

The rate of the trigger signal must be the same as the rate of the data signal input. You specify the triggering event for the Push, Pop, and Clr ports by the **Trigger type** pop-up menu:

- Rising edge – Triggers execution of the block when the trigger input does one of the following:

  - Rises from a negative value to a positive value or zero

  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero; see the following figure

- Falling edge – Triggers execution of the block when the trigger input does one of the following:

  - Falls from a positive value to a negative value or zero

  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero; see the following figure

- Either edge – Triggers execution of the block when the trigger input is a Rising edge or Falling edge (as described above).

- Non-zero sample – Triggers execution of the block at each sample time that the trigger input is not zero.

Note When running simulations in the Simulink MultiTasking mode, sample-based trigger signals have a one-sample latency, and frame-based trigger signals have one frame of latency. Thus, there is a one-sample or one-frame delay between the time the block detects a trigger event, and when it applies the trigger. For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" on page 2-56 and "Models with Multiple Sample Rates" in the Real-Time Workshop User's Guide documentation.

Queue

The **Push onto full register** parameter specifies the block's behavior when a trigger is received at the Push port but the register is full. The **Pop empty register** parameter specifies the block's behavior when a trigger is received at the Pop port but the register is empty. The following options are available for both cases:

- Ignore – Ignore the trigger event, and continue the simulation.

- Warning – Ignore the trigger event, but display a warning message in the MATLAB Command Window.

- Error – Display an error dialog box and terminate the simulation.

> **Note** The **Push onto full register** and **Pop empty register** parameters are diagnostic parameters. Like all diagnostic parameters on the Configuration Parameters dialog box, they are set to Ignore in the Real-Time Workshop code generated for this block.

The **Push onto full register** parameter additionally offers the **Dynamic reallocation** option, which dynamically resizes the register to accept as many additional inputs as memory permits. To find out how many elements are on the queue at a given time, enable the Num output port by selecting the **Output number of register entries** option.

Examples

Example 1

The table below illustrates the Queue block's operation for a **Register size** of 4, **Trigger type** of Either edge, and **Clear output port on reset** enabled. Because the block triggers on both rising and falling edges in this example, each transition from 1 to 0 or 0 to 1 in the Push, Pop, and Clr columns below represents a distinct trigger event. A 1 in the Empty column indicates an empty queue, while a 1 in the Full column indicates a full queue.

In	Push	Pop	Clr	Queue	Out	Empty	Full	Num
1	0	0	0	top ▮ ▯ ▯ ▯ ▯ ▮ bottom	0	1	0	0
2	1	0	0	top ▮ ▯ ▯ ▯ [2] ▮ bottom	0	0	0	1
3	0	0	0	top ▮ ▯ ▯ [3] [2] ▮ bottom	0	0	0	2
4	1	0	0	top ▮ ▯ [4] [3] [2] ▮ bottom	0	0	0	3
5	0	0	0	top ▮ [5] [4] [3] [2] ▮ bottom	0	0	1	4
6	0	1	0	top ▮ ▯ [5] [4] [3] ▮ bottom	2	0	0	3
7	0	0	0	top ▮ ▯ ▯ [5] [4] ▮ bottom	3	0	0	2
8	0	1	0	top ▮ ▯ ▯ ▯ [5] ▮ bottom	4	0	0	1
9	0	0	0	top ▮ ▯ ▯ ▯ ▯ ▮ bottom	5	1	0	0
10	1	0	0	top ▮ ▯ ▯ ▯ [10] ▮ bottom	5	0	0	1
11	0	0	0	top ▮ ▯ ▯ [11] [10] ▮ bottom	5	0	0	2
12	1	0	1	top ▮ ▯ ▯ ▯ [12] ▮ bottom	0	0	0	1

Note that at the last step shown, the Push and Clr ports are triggered simultaneously. The Clr trigger takes precedence, and the queue is first cleared and then pushed.

Example 2

The dspqdemo demo provides another example of the operation of the Queue block.

Queue

Dialog Box
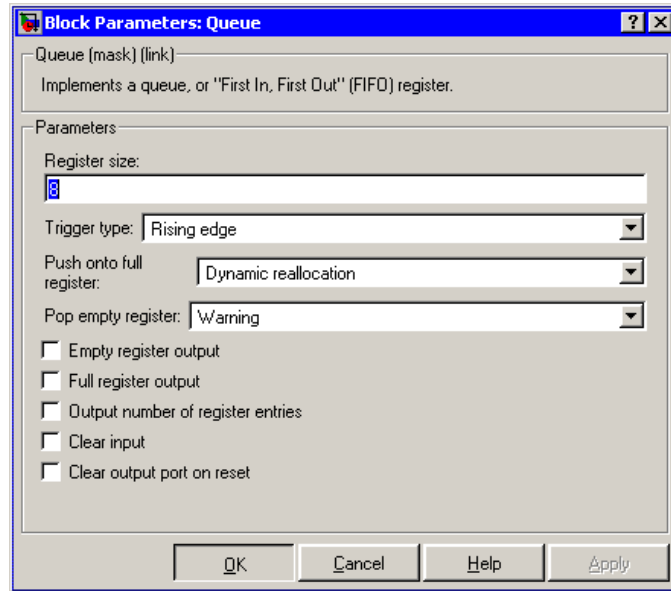
Register size
> The number of entries that the FIFO register can hold.

Trigger type
> The type of event that triggers the block's execution. The rate of the trigger signal must be the same as the rate of the data signal input. Tunable.

Push onto full register
> Response to a trigger received at the Push port when the register is full. Inputs to this port must have the same built-in data type as inputs to the Pop and Clr input ports.

Pop empty register
> Response to a trigger received at the Pop port when the register is empty. Inputs to this port must have the same built-in data type as inputs to the Push and Clr input ports. Tunable.

Empty register output

Enable the Empty output port, which is high (1) when the queue is empty, and low (0) otherwise.

Full register output

Enable the Full output port, which is high (1) when the queue is full, and low (0) otherwise. The Full port remains low when you select Dynamic reallocation from the **Push onto full register** parameter.

Output number of register entries

Enable the Num output port, which tracks the number of entries currently on the queue. When inputs to the In port are double-precision values, the outputs from the Num port are double-precision values. Otherwise, the outputs from the Num port are 32-bit unsigned integer values.

Clear input

Enable the Clr input port, which empties the queue when the trigger specified by the **Trigger type** is received. Inputs to this port must have the same built-in data type as inputs to the Push and Pop input ports.

Clear output port on reset

Reset the Out port to zero, in addition to clearing the queue, when a trigger is received at the Clr input port. Tunable.

Queue

**Supported
Data
Types**

Port	Supported Data Types
In	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Custom data types
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
Push	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Custom data types
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
	Inputs to this port must have the same built-in data type as inputs to the Pop and Clr input ports

Port	Supported Data Types
Pop	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Custom data types
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
	Inputs to this port must have the same built-in data type as inputs to the Push and Clr input ports.
Clr	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed only)
	• Custom data types
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
	Inputs to this port must have the same built-in data type as inputs to the Push and Pop input ports.

Queue

Port	Supported Data Types
Out	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • Custom data types • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Empty	• Double-precision floating point • Boolean The block outputs Boolean values at this port when Boolean support is enabled, as described in "Effects of Enabling and Disabling Boolean Support" on page 7-17. To learn how to disable Boolean output support, see "Steps to Disabling Boolean Support" on page 7-18
Full	• Double-precision floating point • Boolean The block outputs Boolean values at this port when Boolean support is enabled, as described in "Effects of Enabling and Disabling Boolean Support" on page 7-17. To learn how to disable Boolean output support, see "Steps to Disabling Boolean Support" on page 7-18
Num	• Double-precision floating point The block outputs a double-precision floating-point value at this port when the data type of the In port is double-precision floating-point. • 32-bit unsigned integers

Port	Supported Data Types
	The block outputs a 32-bit unsigned integer value at this port when the data type of the In port is anything other than double-precision floating-point.

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Buffer	Signal Processing Blockset
Delay Line	Signal Processing Blockset
Stack	Signal Processing Blockset

Random Source

Purpose   Generate randomly distributed values

Library   Signal Processing Sources

Description

The Random Source block generates a frame of M values drawn from a uniform or Gaussian pseudorandom distribution, where you specify M in the **Samples per frame** parameter.

This reference page contains a detailed discussion of the following Random Source block topics:

- "Distribution Type" on page 10-860
- "Output Complexity" on page 10-861
- "Output Repeatability" on page 10-863
- "Specifying the Initial Seed" on page 10-863
- "Sample Period" on page 10-864
- "Dialog Box" on page 10-865
- "Supported Data Types" on page 10-868
- "See Also" on page 10-869

Distribution Type

When the **Source type** parameter is set to `Uniform`, the output samples are drawn from a uniform distribution whose minimum and maximum values are specified by the **Minimum** and **Maximum** parameters, respectively. All values in this range are equally likely to be selected. A length-N vector specified for one or both of these parameters generates an N-channel output (M-by-N matrix) containing a unique random distribution in each channel.

For example, specify

- **Minimum** = [0 0 -3 -3]
- **Maximum** = [10 10 20 20]

to generate a four-channel output whose first and second columns contain random values in the range [0, 10], and whose third and fourth columns contain random values in the range [-3, 20]. When you specify only one of the **Minimum** and **Maximum** parameters as a vector, the block scalar expands the other parameter so it is the same length as the vector.

When the **Source type** parameter is set to Gaussian, you must also set the **Method** parameter, which determines the method by which the block computes the output, and has the following settings:

- Ziggurat – Produces Gaussian random values by using the Ziggurat method, which is the same method used by the MATLAB randn function.

- Sum of uniform values – Produces Gaussian random values by adding and scaling uniformly distributed random signals based on the central limit theorem. This theorem states that the probability distribution of the sum of a sufficiently high number of random variables approaches the Gaussian distribution. You must set the **Number of uniform values to sum** parameter, which determines the number of uniformly distributed random numbers to sum to produce a single Gaussian random value.

For both settings of the **Method** parameter, the output samples are drawn from the normal distribution defined by the **Mean** and **Variance** parameters. A length-N vector specified for one or both of the **Mean** and **Variance** parameters generates an N-channel output (M-by-N frame matrix) containing a distinct random distribution in each column. When you specify only one of these parameters as a vector, the block scalar expands the other parameter so it is the same length as the vector.

Output Complexity

The block's output can be either real or complex, as determined by the Real and Complex options in the **Complexity** parameter. These settings control all channels of the output, so real and complex data cannot be combined in the same output. For complex output with a Uniform distribution, the real and imaginary components in each

channel are both drawn from the same uniform random distribution, defined by the **Minimum** and **Maximum** parameters for that channel.

For complex output with a `Gaussian` distribution, the real and imaginary components in each channel are drawn from normal distributions with different means. In this case, the **Mean** parameter for each channel should specify a complex value; the real component of the **Mean** parameter specifies the mean of the real components in the channel, while the imaginary component specifies the mean of the imaginary components in the channel. When either the real or imaginary component is omitted from the **Mean** parameter, a default value of 0 is used for the mean of that component.

For example, a **Mean** parameter setting of `[5+2i 0.5 3i]` generates a three-channel output with the following means.

Channel 1 mean	*real* = 5	*imaginary* = 2
Channel 2 mean	*real* = 0.5	*imaginary* = 0
Channel 3 mean	*real* = 0	*imaginary* = 3

For complex output, the **Variance** parameter, σ^2, specifies the *total variance* for each output channel. This is the sum of the variances of the real and imaginary components in that channel.

$$\sigma^2 = \sigma_{Re}^2 + \sigma_{Im}^2$$

The specified variance is equally divided between the real and imaginary components, so that

$$\sigma_{Re}^2 = \frac{\sigma^2}{2}$$

$$\sigma_{Im}^2 = \frac{\sigma^2}{2}$$

Output Repeatability

The **Repeatability** parameter determines whether or not the block outputs the same signal each time you run the simulation. You can set the parameter to one of the following options:

- Repeatable – Outputs the same signal each time you run the simulation. The first time you run the simulation, the block randomly selects an initial seed. The block reuses these same initial seeds every time you rerun the simulation.

- Specify seed – Outputs the same signal each time you run the simulation. Every time you run the simulation, the block uses the initial seed(s) specified in the **Initial seed** parameter. Also see"Specifying the Initial Seed" on page 10-863.

- Not repeatable – Does not output the same signal each time you run the simulation. Every time you run the simulation, the block randomly selects an initial seed.

Specifying the Initial Seed

When you set the **Repeatability** parameter to Specify seed, you must set the **Initial seed** parameter. The **Initial seed** parameter specifies the initial seed for the pseudorandom number generator. The generator produces an identical sequence of pseudorandom numbers each time it is executed with a particular initial seed.

Specifying Initial Seeds for Real Outputs

To specify the N initial seeds for an N-channel real-valued output, **Complexity** parameter set to Real, provide one of the following in the **Initial seed** parameter:

- Length-N vector of initial seeds – Uses each vector element as an initial seed for the corresponding channel in the N-channel output.

- Single scalar – Uses the scalar to generate N random values, which it uses as the seeds for the N-channel output.

Random Source

Specifying Initial Seeds for Complex Outputs

To specify the initial seeds for an N-channel complex-valued output, **Complexity** parameter set to `Complex`, provide one of the following in the **Initial seed** parameter:

- Length-N vector of initial seeds – Uses each vector element as an initial seed for generating N channels of *real* random values. The block uses pairs of adjacent values in each of these channels as the real and imaginary components of the final output, as illustrated in the following figure.

- Single scalar – Uses the scalar to generate N random values, which it uses as the seeds for generating N channels of *real* random values. The block uses pairs of adjacent values in each of these channels as the real and imaginary components of the final output, as illustrated in the following figure.
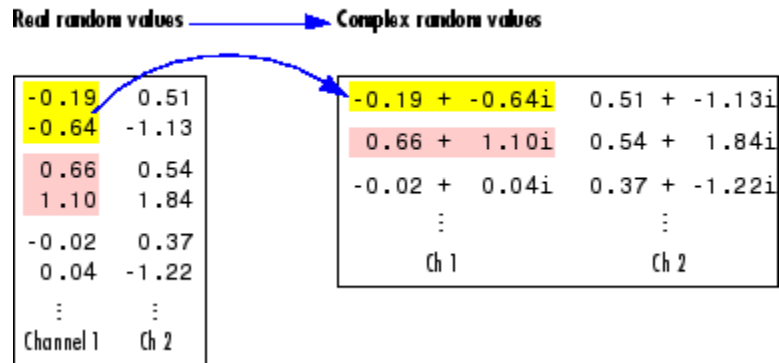
Sample Period

The **Sample time** parameter value, T_s, specifies the random sequence sample period when the **Sample mode** parameter is set to `Discrete`. In this mode, the block generates the number of samples specified by the **Samples per frame** parameter value, M, and outputs this frame

with a period of $M*T_s$. For M=1, the output is sample based; otherwise, the output is frame based.

When **Sample mode** is set to Continuous, the block is configured for continuous-time operation, and the **Sample time** and **Samples per frame** parameters are disabled. Note that many blocks in the DSP Blockset do not accept continuous-time inputs.

Dialog Box

Only some of the parameters described below are visible in the dialog box at any one time.
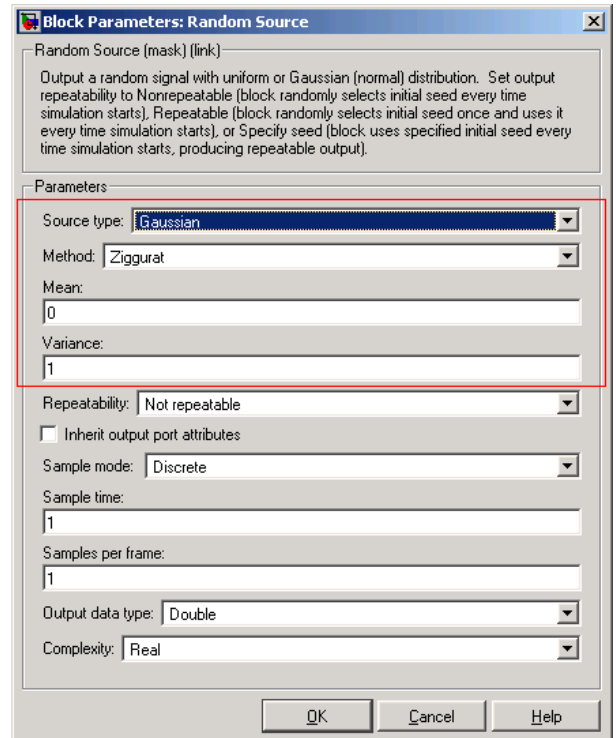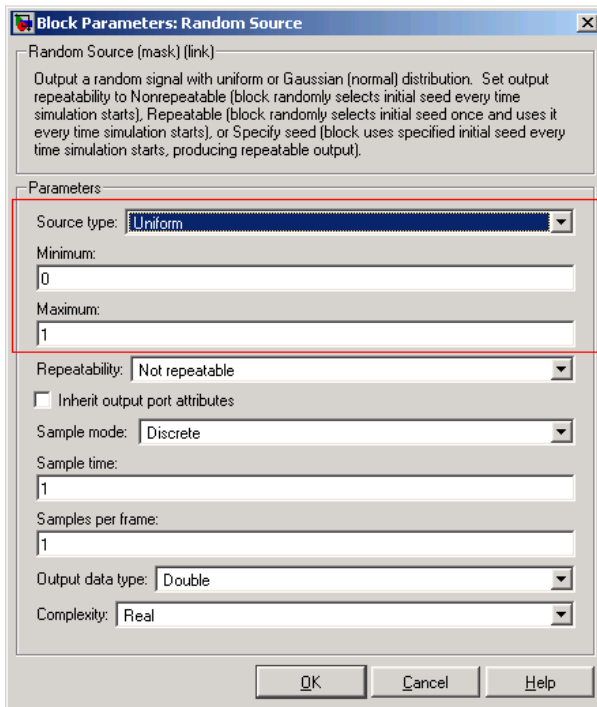
Random Source

Opening this dialog box causes a running simulation to pause. See "Changing Source Block Parameters" in the online Simulink documentation for details.

Source type
> The distribution from which to draw the random values, Uniform or Gaussian. For more information, see "Distribution Type" on page 10-860.

Method
> The method by which the block computes the Gaussian random values, Ziggurat or Sum of uniform values. This parameter is enabled when **Source type** is set to Gaussian. For more information, see "Distribution Type" on page 10-860.

Minimum
> The minimum value in the uniform distribution. This parameter is enabled when you select Uniform from the **Source type** parameter. Tunable.

Maximum
> The maximum value in the uniform distribution. This parameter is enabled when you select you select Uniform from the **Source type** parameter. Tunable.

Number of uniform values to sum
> The number of uniformly distributed random values to sum to compute a single number in a Gaussian random distribution. This parameter is enabled when the **Source type** parameter is set to Gaussian, and the **Method** parameter is set to Sum of uniform values. For more information, see "Distribution Type" on page 10-860.

Mean
> The mean of the Gaussian (normal) distribution. This parameter is enabled when you select Gaussian from the **Source type** parameter. Tunable.

Variance

The variance of the Gaussian (normal) distribution. This parameter is enabled when you select `Gaussian` from the **Source type** parameter. Tunable.

Repeatability

The repeatability of the block output: `Not repeatable`, `Repeatable`, or `Specify seed`. In the `Repeatable` and `Specify seed` settings, the block outputs the same signal every time you run the simulation. For details, see "Output Repeatability" on page 10-863.

Initial seed

The initial seed(s) to use for the random number generator when you set the **Repeatability** parameter to `Specify seed`. For details, see "Specifying the Initial Seed" on page 10-863. Tunable.

Inherit output port attributes

When you select this check box, block inherits the sample mode, sample time, output data type, complexity, and signal dimensions of a sample-based signal from a downstream block. When you select this check box, the **Sample mode**, **Sample time**, **Samples per frame**, **Output data type**, and **Complexity** parameters are disabled.

Suppose you want to back propagate a 1-D vector. The output of the Random Source block is a length M sample-based 1-D vector, where length M is inherited from the downstream block. When the **Minimum**, **Maximum**, **Mean**, or **Variance** parameter specifies N channels, the 1-D vector output contains M/N samples from each channel. An error occurs in this case when M is not an integer multiple of N.

Suppose you want to back propagate a M-by-N signal. When N>1, your signal has N channels. When N = 1, your signal has M channels. The value of the **Minimum**, **Maximum**, **Mean**, or **Variance** parameter can be a scalar or a vector of length equal to the number of channels. You can specify these parameters

as either row or column vectors, except when the signal is a row vector. In this case, the **Minimum**, **Maximum**, **Mean**, or **Variance** parameter must also be specified as a row vector.

Sample mode

The sample mode, Continuous or Discrete. This parameter is enabled when the **Inherit output port attributes** check box is cleared.

Sample time

The sample period, T_s, of the random output sequence. The output frame period is $M*T_s$. This parameter is enabled when the **Inherit output port attributes** check box is cleared.

Samples per frame

The number of samples, M, in each output frame. This parameter is enabled when the **Inherit output port attributes** check box is cleared.

Output data type

The data type of the output, single-precision or double-precision. This parameter is enabled when the **Inherit output port attributes** check box is cleared.

Output complexity

The complexity of the output, Real or Complex. This parameter is enabled when the **Inherit output port attributes** check box is cleared.

Supported Data Types

- Double-precision floating-point
- Single-precision floating-point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Discrete Impulse	Signal Processing Blockset
DSP Constant	Signal Processing Blockset
Maximum	Signal Processing Blockset
Minimum	Signal Processing Blockset
Signal From Workspace	Signal Processing Blockset
Standard Deviation	Signal Processing Blockset
Variance	Signal Processing Blockset
Random Number	Simulink
Signal Generator	Simulink
rand	MATLAB
randn	MATLAB

Real Cepstrum

Purpose　　　Compute real cepstrum of input

Library　　　Transforms

Description　The Real Cepstrum block computes the real cepstrum of each channel in the real-valued M-by-N input matrix, u. For both sample-based and frame-based inputs, the block assumes that each input column is a frame containing M consecutive samples from an independent channel. The block does not accept complex-valued inputs.

The output is a real M_o-by-N matrix, where you specify M_o in the **FFT length** parameter. Each output column contains the length-M_o real cepstrum of the corresponding input column.

```
y = real(ifft(log(abs(fft(u,Mo)))))    % Equivalent MATLAB code
```

or, more compactly,

```
y = rceps(u,Mo)
```

When you select the **Inherit FFT length from input port dimensions** check box, the output frame size matches the input frame size (M_o=M). In this case, a *sample-based* length-M row vector input is processed as a single channel, that is, as an M-by-1 column vector, and the output is a length-M row vector. A 1-D vector input is *always* processed as a single channel, and the output is a 1-D vector.

The output is always sample based, and the output port rate is the same as the input port rate.

Dialog Box

Inherit FFT length from input port dimensions
When selected, matches the output frame size to the input frame size.

FFT length
The number of frequency points at which to compute the FFT, which is also the output frame size, M_o. This parameter is available when you do not select **Inherit FFT length from input port dimensions**.

Supported Data Types
- Double-precision floating point
- Single-precision floating point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Complex Cepstrum	Signal Processing Blockset
DCT	Signal Processing Blockset
FFT	Signal Processing Blockset
rceps	Signal Processing Toolbox

Reciprocal Condition

Purpose   Compute reciprocal condition of square matrix in the 1-norm

Library   Math Functions / Matrices and Linear Algebra / Matrix Operations

Description   The Reciprocal Condition block computes the reciprocal of the condition
number for a square input matrix A.

```
y = rcond(A)      % Equivalent MATLAB code
```

or

$$y = \frac{1}{\kappa} = \frac{1}{\left\|A^{-1}\right\|_1 \left\|A\right\|_1}$$

where κ is the condition number ($\kappa \geq 1$), and y is the scalar sample-based
output ($0 \leq y < 1$).

The matrix 1-norm, $\left\|A\right\|_1$, is the maximum column-sum in the M-by-M
matrix A.

$$\left\|A\right\|_1 = \max_{1 \leq j \leq M} \sum_{i=1}^{M} \left|a_{ij}\right|$$

For a 3-by-3 matrix:
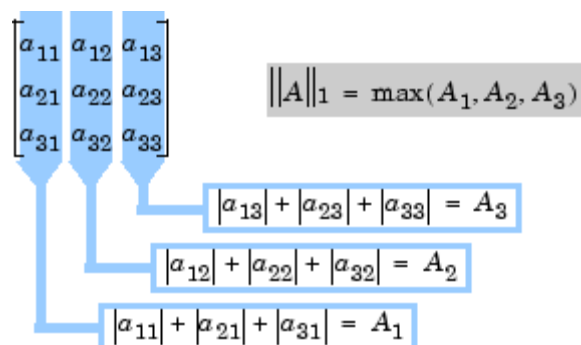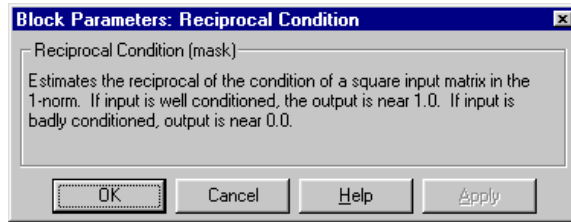
$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \qquad \left\|A\right\|_1 = \max(A_1, A_2, A_3)$$

$$\left|a_{13}\right| + \left|a_{23}\right| + \left|a_{33}\right| = A_3$$

$$\left|a_{12}\right| + \left|a_{22}\right| + \left|a_{32}\right| = A_2$$

$$\left|a_{11}\right| + \left|a_{21}\right| + \left|a_{31}\right| = A_1$$

Dialog Box


```
Block Parameters: Reciprocal Condition                    ☒
─ Reciprocal Condition (mask) ──────────────────────────
  Estimates the reciprocal of the condition of a square input matrix in the
  1-norm.  If input is well conditioned, the output is near 1.0.  If input is
  badly conditioned, output is near 0.0.


        OK          Cancel          Help          Apply
```

References   Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

Supported Data Types
- Double-precision floating point
- Single-precision floating point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Matrix 1-Norm	Signal Processing Blockset
Normalization	Signal Processing Blockset
rcond	MATLAB

Remez FIR Filter Design

Purpose   Design and apply an equiripple FIR filter.

Library   dspobslib

Description

Note The Remez FIR Filter Design block is still supported but is likely to be obsoleted in a future release. We strongly recommend replacing this block with the Digital Filter block.

The Remez FIR Filter Design block implements the Parks-McClellan algorithm to design and apply a linear-phase filter with an arbitrary multiband magnitude response. The filter design, which uses the firpm function in the Signal Processing Toolbox, minimizes the maximum error between the desired frequency response and the actual frequency response. Such filters are called *equiripple* due to the equiripple behavior of their approximation error. The block applies the filter to a discrete-time input using the Direct-Form II Transpose Filter block.

An M-by-N sample-based matrix input is treated as M*N independent channels, and an M-by-N frame-based matrix input is treated as N independent channels. In both cases, the block filters each channel independently over time, and the output has the same size and frame status as the input.

The **Filter type** parameter allows you to specify one of the following filters:

- Multiband

  The multiband filter has an arbitrary magnitude response and linear phase.

- Differentiator

  The differentiator filter approximates the ideal differentiator. Differentiators are antisymmetric FIR filters with approximately linear magnitude responses. To obtain the correct derivative, scale

the **Gains at these frequencies** vector by πF_s rad/s, where F_s is the sample frequency in Hertz.

- `Hilbert Transformer`

  The Hilbert transformer filter approximates the ideal Hilbert transformer. Hilbert transformers are antisymmetric FIR filters with approximately constant magnitude.

The **Band-edge frequency vector** parameter is a vector of frequency points in the range 0 to 1, where 1 corresponds to half the sample frequency. Each band is defined by the two bounding frequencies, so this vector must have even length. Frequency points must appear in ascending order. The **Gains at these frequencies** parameter is a vector of the same size containing the desired magnitude response at the corresponding points in the **Band-edge frequency vector**.

Each odd-indexed frequency-amplitude pair defines the left endpoint of a line segment representing the desired magnitude response in that frequency band. The corresponding even-indexed frequency-amplitude pair defines the right endpoint. Between the frequency bands specified by these end-points, there may be undefined sections of the specified frequency response. These are called "don't care" or "transition" regions, and the magnitude response in these areas is a by-product of the optimization in the other specified frequency ranges.

Remez FIR Filter Design
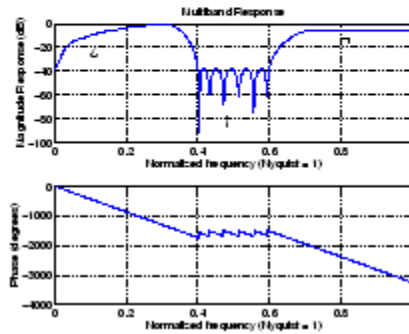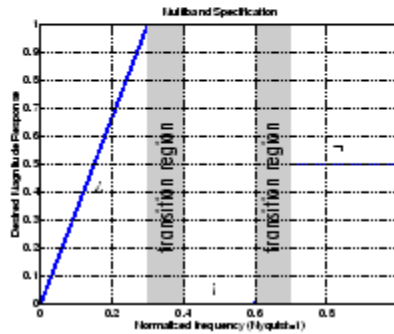
Band edge frequency = [0 0.3 0.4 0.6 0.7 1]

Gains = [0 1 0 0 0.5 0.5]

Band:

The **Weights** parameter is a vector that specifies the emphasis to be placed on minimizing the error in certain frequency bands relative to others. This vector specifies one weight per band, so it is half the length of the **Band-edge frequency vector** and **Gains at these frequencies** vectors.

In most cases, differentiators and Hilbert transformers have only a single band, so the weight is a scalar value that does not affect the final filter. However, the **Weights** parameter is useful when using the block to design an antisymmetric multiband filter, such as a Hilbert transformer with stopbands.

Examples

Example 1: Multiband

Consider a lowpass filter with a transition band in the normalized frequency range 0.4 to 0.5, and 10 times greater error minimization in the stopband than in the passband.

In this case:

- **Filter type** = Multiband

- **Band-edge frequency vector** = [0 0.4 0.5 1]

- **Gains at these frequencies** = [1 1 0 0]

- **Weights** = [1 10]

Example 2: Differentiator

Assume the specifications for a differentiator filter require it to have order 21. The "ramp" response extends over the entire frequency range. In this case, specify:

- **Filter type** = Differentiator

- **Band-edge frequency vector** = [0 1]

- **Gains at these frequencies** = [0 pi*Fs]

- **Filter order** = 21

For a type III even order filter, the differentiation band should stop short of half the sample frequency. For example, if the filter order is 20, you could specify the block parameters as follows:

- **Filter type** = Differentiator

- **Band-edge frequency vector** = [0 0.9]

- **Gains at these frequencies** = [0 0.9*pi*Fs]

- **Filter order** = 20

Remez FIR Filter Design

**Dialog
Box**
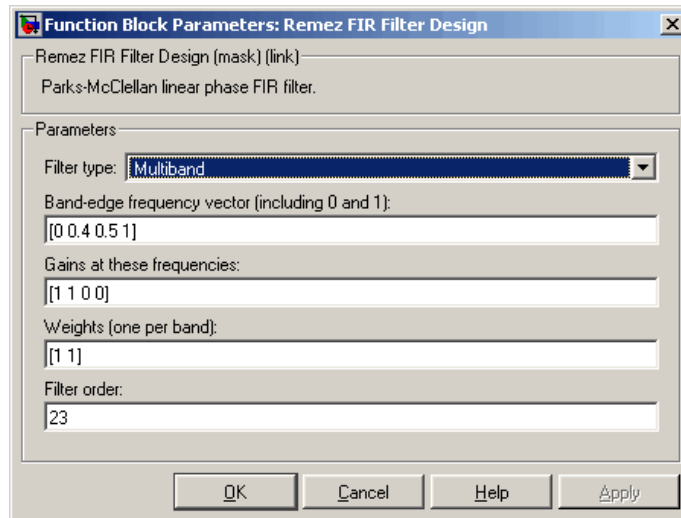
Filter type
The filter type. Tunable.

Band-edge frequency vector
A vector of frequency points, in ascending order, in the range
0 to 1. The value 1 corresponds to half the sample frequency. This
vector must have even length. Tunable.

Gains at these frequencies
A vector of frequency-response magnitudes corresponding to the
points in the **Band-edge frequency** vector. This vector must be
the same length as the **Band-edge frequency** vector. Tunable.

Weights
A vector containing one weight for each frequency band. This
vector must be half the length of the **Band-edge frequency** and
Gains at these frequencies vectors. Tunable.

Filter order
The filter order.

References   Oppenheim, A. V. and R. W. Schafer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989.

Proakis, J. and D. Manolakis. *Digital Signal Processing.* 3rd ed. Englewood Cliffs, NJ: Prentice-Hall, 1996.

Repeat

Purpose   Resample input at higher rate by repeating values

Library   Signal Operations

Description   The Repeat block upsamples each channel of the M_i-by-N input to a rate L times higher than the input sample rate by repeating each consecutive input sample L times at the output. You specify the integer L in the **Repetition count** parameter.

This block supports triggered subsystems if, for **Frame-based mode,** you select `Maintain input frame rate`.

Sample-Based Operation

When the input is sample based, the block treats each of the M*N matrix elements as an independent channel, and upsamples each channel over time. The **Frame-based mode** parameter must be set to `Maintain input frame size`. The output sample rate is L times higher than the input sample rate ($T_{so} = T_{si}/L$), and the input and output sizes are identical.

Frame-Based Operation

When the input is frame based, the block treats each of the N input columns as a frame containing M_i sequential time samples from an independent channel. The block upsamples each channel independently by repeating each row of the input matrix L times at the output. The **Frame-based mode** parameter determines how the block adjusts the rate at the output to accommodate the repeated rows. There are two available options:

• `Maintain input frame size`

  The block generates the output at the faster (upsampled) rate by using a proportionally shorter frame *period* at the output port than at the input port. For L repetitions of the input, the output frame period is L times shorter than the input frame period ($T_{fo} = T_{fi}/L$), but the input and output frame sizes are equal.

The model below shows a single-channel input with a frame period of 1 second being upsampled through 4-times repetition to a frame period of 0.25 second. The input and output frame sizes are identical.
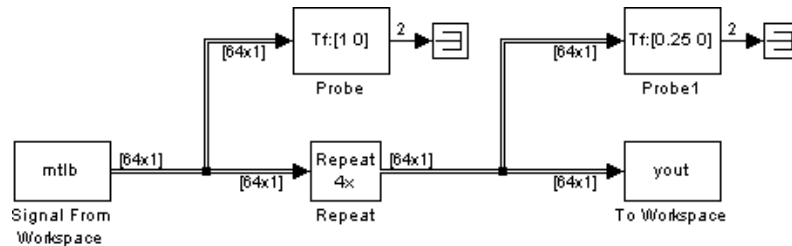
- Maintain input frame rate

  The block generates the output at the faster (upsampled) rate by using a proportionally larger frame *size* than the input. For L repetitions of the input, the output frame size is L times larger than the input frame size ($M_o = M_i*L$), but the input and output frame rates are equal.

  The model below shows a single-channel input of frame size 16 being upsampled through 4-times repetition to a frame size of 64. The input and output frame rates are identical.
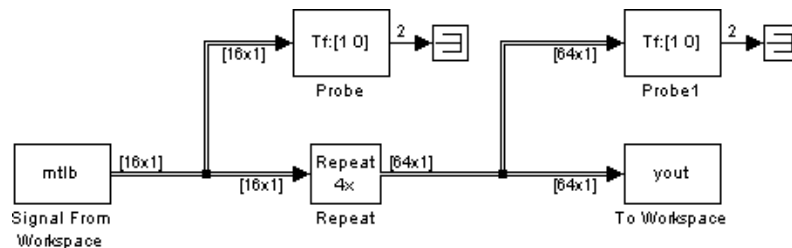
Zero Latency

The Repeat block has *zero-tasking latency* for all single-rate operations. The block is single-rate for the particular combinations of sampling mode and parameter settings shown in the table below.

Repeat

Sampling Mode	Parameter Settings
Sample based	**Repetition count** parameter, L, is 1.
Frame based	**Repetition count** parameter, L, is 1, *or*
	Frame-based mode parameter is `Maintain input frame rate`.

The block also has zero latency for all multirate operations in the Simulink single-tasking mode.

Zero tasking latency means that the block repeats the first input (received at t=0) for the first L output samples, the second input for the next L output samples, and so on. The **Initial condition** parameter value is not used.

Nonzero Latency

The Repeat block has tasking latency only for multirate operation in the Simulink multitasking mode:

- In sample-based mode, the initial condition for each channel is repeated for the first L output samples. The channel's first input appears as output sample L+1. The **Initial condition** value can be an M_i-by-N matrix containing one value for each channel, or a scalar to be applied to all signal channels.

- In frame-based mode, the first row of the initial condition matrix is repeated for the first L output samples, the second row of the initial condition matrix is repeated for the next L output samples, and so on. The first row of the first input matrix appears in the output as sample M_iL+1. The **Initial condition** value can be an M_i-by-N matrix, or a scalar to be repeated across all elements of the M_i-by-N matrix. See the example below for an illustration of this case.

Note For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" on page 2-56 and "Models with Multiple Sample Rates" in the Real-Time Workshop User's Guide documentation.

Examples   Construct the frame-based model shown below.

Adjust the block parameters as follows.

- Configure the Signal From Workspace block to generate a two-channel signal with frame size of 4 and sample period of 0.25. This represents an output frame period of 1 (0.25*4). The first channel should contain the positive ramp signal 1, 2, ..., 100, and the second channel should contain the negative ramp signal -1, -2, ..., -100.

  - **Signal** = [(1:100)' (-1:-1:-100)']

  - **Sample time** = 0.25

  - **Samples per frame** = 4

- Configure the Repeat block to upsample the two-channel input by increasing the output frame rate by a factor of 2 relative to the input frame rate. Set an initial condition matrix of

$$\begin{bmatrix} 11 & -11 \\ 12 & -12 \\ 13 & -13 \\ 14 & -14 \end{bmatrix}$$

- **Repetition count** = 2
- **Initial condition** = [11 -11;12 -12;13 -13;14 -14]
- **Frame-based mode** = Maintain input frame size

- Configure the Probe blocks by clearing the **Probe width** and **Probe complex signal** check boxes (if desired).

This model is multirate because there are at least two distinct sample rates, as shown by the two Probe blocks. To run this model in the Simulink multitasking mode, in the **Solver** pane of the Configuration Parameters dialog box, set the **Type** list to Fixed-step and set the **Solver** list to discrete (no continuous states). For the **Tasking mode for periodic sample times** parameter, select MultiTasking. Also set the **Stop time** to 30.

Run the model and look at the output, yout. The first few samples of each channel are shown below.

```
yout =
    11    -11
    11    -11
    12    -12
    12    -12
    13    -13
    13    -13
    14    -14
    14    -14
     1     -1
     1     -1
     2     -2
     2     -2
     3     -3
```

```
3     -3
4     -4
4     -4
5     -5
5     -5
```

Since we ran this frame-based multirate model in multitasking mode, the block repeats each row of the initial condition matrix for L output samples, where L is the **Repetition count** of 2. The first row of the first input matrix appears in the output as sample 9 ,that is, sample M_iL+1, where M_i is the input frame size.

Dialog Box
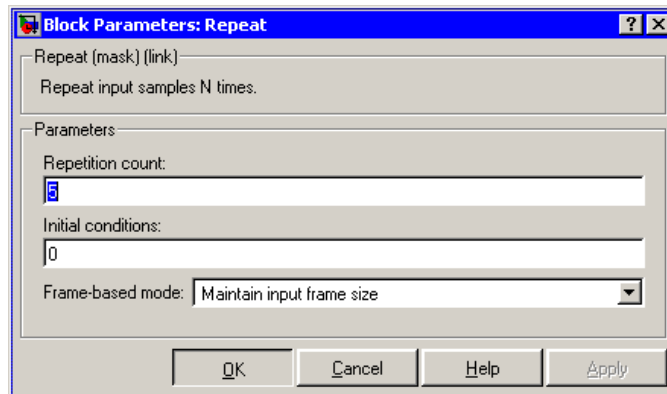
Repetition count

The integer number of times, L, that the input value is repeated at the output. This is the factor by which the output frame size or sample rate is increased.

Initial conditions

The value with which the block is initialized for cases of nonzero latency; a scalar or matrix.

Frame-based mode

For frame-based operation, the method by which to implement the repetition (upsampling): Maintain input frame size that

is, increase the frame rate, or `Maintain input frame rate`, that is, increase the frame size. The **Frame-based mode** parameter must be set to `Maintain input frame size` for sample-base inputs.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Custom data types
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Custom data types
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

FIR Interpolation	Signal Processing Blockset
Upsample	Signal Processing Blockset
Zero Pad	Signal Processing Blockset

RLS Adaptive Filter

Purpose　　Compute filter estimates for an input using the RLS adaptive filter algorithm

Library　　dspobslib

Description
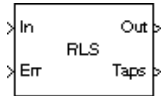
> **Note** The RLS Adaptive Filter block is still supported but is likely to be obsoleted in a future release. We recommend replacing this block with the RLS Filter block.

The RLS Adaptive Filter block recursively computes the recursive least squares (RLS) estimate of the FIR filter coefficients.

The corresponding RLS filter is expressed in matrix form as

$$k(n) = \frac{\lambda^{-1}P(n-1)u(n)}{1+\lambda^{-1}u^H(n)P(n-1)u(n)}$$

$$y(n) = \hat{w}^H(n-1)u(n)$$

$$e(n) = d(n) - y(n)$$

$$\hat{w}(n) = \hat{w}(n-1) + k(n)e^*(n)$$

$$P(n) = \lambda^{-1}P(n-1) - \lambda^{-1}k(n)u^H(n)P(n-1)$$

where λ^{-1} denotes the reciprocal of the exponential weighting factor. The variables are as follows

Variable	Description
n	The current algorithm iteration
$u(n)$	The buffered input samples at step n
$P(n)$	The inverse correlation matrix at step n
$k(n)$	The gain vector at step n
$\hat{w}(n)$	The vector of filter-tap estimates at step n

Variable	Description
$y(n)$	The filtered output at step n
$e(n)$	The estimation error at step n
$d(n)$	The desired response at step n
λ	The exponential memory weighting factor

The block icon has port labels corresponding to the inputs and outputs of the RLS algorithm. Note that inputs to the In and Err ports must be sample-based scalars. The signal at the Out port is a scalar, while the signal at the Taps port is a sample-based vector.

Block Ports	Corresponding Variables
In	u, the scalar input, which is internally buffered into the vector $u(n)$
Out	$y(n)$, the filtered scalar output
Err	$e(n)$, the scalar estimation error
Taps	$\hat{w}(n)$, the vector of filter-tap estimates

An optional Adapt input port is added when you select the **Adapt input** check box in the dialog box. When this port is enabled, the block continuously adapts the filter coefficients while the Adapt input is nonzero. A zero-valued input to the Adapt port causes the block to stop adapting, and to hold the filter coefficients at their current values until the next nonzero Adapt input.

The implementation of the algorithm in the block is optimized by exploiting the symmetry of the inverse correlation matrix $P(n)$. This decreases the total number of computations by a factor of two.

The **FIR filter length** parameter specifies the length of the filter that the RLS algorithm estimates. The **Memory weighting factor** corresponds to λ in the equations, and specifies how quickly the filter

RLS Adaptive Filter

"forgets" past sample information. Setting λ=1 specifies an infinite memory; typically, `0.95` ≤λ≤1.

The **Initial value of filter taps** specifies the initial value $\hat{w}(0)$ as a vector, or as a scalar to be repeated for all vector elements. The initial value of $P(n)$ is

$$I\frac{1}{\hat{\sigma}^2}$$

where you specify $\hat{\sigma}^2$ in the **Initial input variance estimate** parameter.

Examples   The `rlsdemo` demo illustrates a noise cancellation system built around the RLS Adaptive Filter block.

Dialog Box
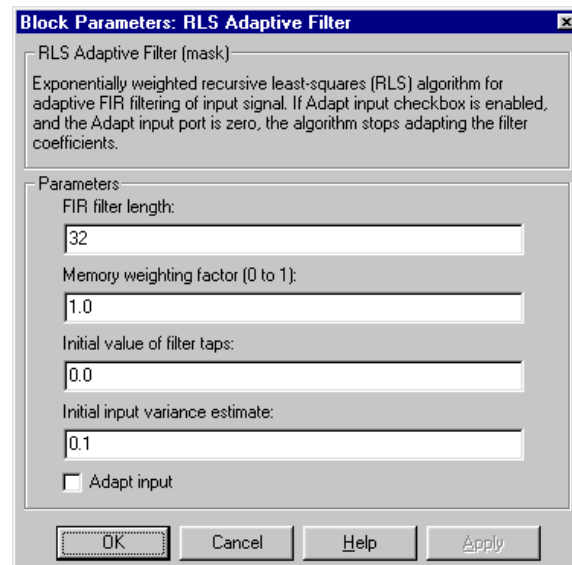
FIR filter length
> The length of the FIR filter.

Memory weighting factor
The exponential weighting factor, in the range [0,1]. A value of 1 specifies an infinite memory. Tunable.

Initial value of filter taps
The initial FIR filter coefficients.

Initial input variance estimate
The initial value of $1/P(n)$.

Adapt input
Enables the Adapt port.

References   Haykin, S. *Adaptive Filter Theory*. 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 1996.

Supported Data Types
- Double-precision floating point
- Single-precision floating point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Kalman Adaptive Filter	Signal Processing Blockset
LMS Adaptive Filter	Signal Processing Blockset

See "Adaptive Filters" on page 3-53 for related information.

RLS Filter

Purpose   Compute filtered output, filter error, and filter weights for a given input and desired signal using RLS adaptive filter algorithm

Library   Filtering / Adaptive Filters

Description
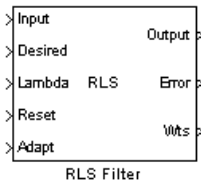
The RLS Filter block recursively computes the least squares estimate (RLS) of the FIR filter weights. The block estimates the filter weights, or coefficients, needed to convert the input signal into the desired signal. Connect the signal you want to filter to the Input port. This input signal can be a sample-based scalar or a single-channel frame-based signal. Connect the signal you want to model to the Desired port. The desired signal must have the same data type, frame status, complexity, and dimensions as the input signal. The Output port outputs the filtered input signal, which can be sample or frame based. The Error port outputs the result of subtracting the output signal from the desired signal.

The corresponding RLS filter is expressed in matrix form as

$$\mathbf{k}(n) = \frac{\lambda^{-1}\mathbf{P}(n-1)\mathbf{u}(n)}{1+\lambda^{-1}\mathbf{u}^{H}(n)\mathbf{P}(n-1)\mathbf{u}(n)}$$

$$y(n) = \mathbf{w}(n-1)\mathbf{u}(n)$$

$$e(n) = d(n) - y(n)$$

$$\mathbf{w}(n) = \mathbf{w}(n-1) + \mathbf{k}^{H}(n)e(n)$$

$$\mathbf{P}(n) = \lambda^{-1}\mathbf{P}(n-1) - \lambda^{-1}\mathbf{k}(n)\mathbf{u}^{H}(n)\mathbf{P}(n-1)$$

where λ^{-1} denotes the reciprocal of the exponential weighting factor. The variables are as follows

Variable	Description
n	The current time index
$\mathbf{u}(n)$	The vector of buffered input samples at step n
$\mathbf{P}(n)$	The inverse correlation matrix at step n

Variable	Description
$\mathbf{k}(n)$	The gain vector at step n
$\mathbf{w}(n)$	The vector of filter-tap estimates at step n
$y(n)$	The filtered output at step n
$e(n)$	The estimation error at step n
$d(n)$	The desired response at step n
λ	The forgetting factor

The implementation of the algorithm in the block is optimized by exploiting the symmetry of the inverse correlation matrix $P(n)$. This decreases the total number of computations by a factor of two.

Use the **Filter length** parameter to specify the length of the filter weights vector.

The **Forgetting factor (0 to 1)** parameter corresponds to λ in the equations. It specifies how quickly the filter "forgets" past sample information. Setting $\lambda=1$ specifies an infinite memory. Typically,

$1 - \dfrac{1}{2L} < \lambda < 1$, where L is the filter length. You can specify a forgetting factor using the input port, Lambda, or enter a value in the **Forgetting factor (0 to 1)** parameter in the Block Parameters: RLS Filter dialog box.

Enter the initial filter weights, $\hat{w}(0)$, as a vector or a scalar for the **Initial value of filter weights** parameter. When you enter a scalar, the block uses the scalar value to create a vector of filter weights. This vector has length equal to the filter length and all of its values are equal to the scalar value.

The initial value of $P(n)$ is

$$\frac{1}{\sigma^2}I$$

where you specify σ^2 in the **Initial input variance estimate** parameter.

When you select the **Adapt port** check box, an Adapt port appears on the block. When the input to this port is nonzero, the block continuously updates the filter weights. When the input to this port is zero, the filter weights remain at their current values.

When you want to reset the value of the filter weights to their initial values, use the **Reset input** parameter. The block resets the filter weights whenever a reset event is detected at the Reset port. The reset signal rate must be the same rate as the data signal input.

From the **Reset input** list, select None to disable the Reset port. To enable the Reset port, select one of the following from the **Reset input** list:

- Rising edge – Triggers a reset operation when the Reset input does one of the following:

  - Rises from a negative value to a positive value or zero

  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero; see the following figure

- Falling edge – Triggers a reset operation when the Reset input does one of the following:

  - Falls from a positive value to a negative value or zero

- Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero; see the following figure

- `Either edge` – Triggers a reset operation when the Reset input is a `Rising edge` or `Falling edge`, as described above

- `Non-zero sample` – Triggers a reset operation at each sample time that the Reset input is not zero

Note When running simulations in the Simulink `MultiTasking` mode, sample-based reset signals have a one-sample latency, and frame-based reset signals have one frame of latency. Thus, there is a one-sample or one-frame delay between the time the block detects a reset event, and when it applies the reset. For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" on page 2-56 and "Models with Multiple Sample Rates" in the Real-Time Workshop User's Guide documentation.

Select the **Output filter weights** check box to create a Wts port on the block. For each iteration, the block outputs the current updated filter weights from this port.

RLS Filter

Examples   The `rlsdemo` demo illustrates a noise cancellation system built around the RLS Filter block.

Dialog Box
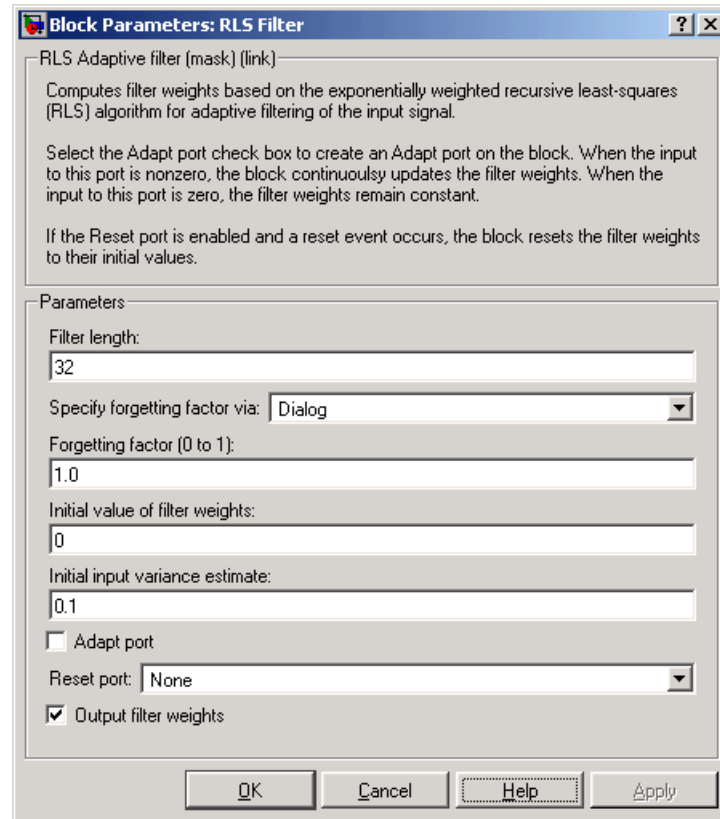
Filter length

    Enter the length of the FIR filter weights vector.

Specify forgetting factor via

    Select `Dialog` to enter a value for the forgetting factor in the Block parameters: RLS Filter dialog box. Select `Input port` to specify the forgetting factor using the Lambda input port.

Forgetting factor (0 to 1)
Enter the exponential weighting factor in the range $0 \leq \lambda \leq 1$. A value of 1 specifies an infinite memory. Tunable.

Initial value of filter weights
Specify the initial values of the FIR filter weights.

Initial input variance estimate
The initial value of $1/P(n)$.

Adapt port
Select this check box to enable the Adapt input port.

Reset input
Select this check box to enable the Reset input port.

Output filter weights
Select this check box to export the filter weights from the Wts port.

References   Hayes, M.H. *Statistical Digital Signal Processing and Modeling*. New York: John Wiley & Sons, 1996.

Supported Data Types

- Double-precision floating point

- Single-precision floating point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Kalman Adaptive Filter	Signal Processing Blockset
LMS Filter	Signal Processing Blockset
Block LMS Filter	Signal Processing Blockset
Fast Block LMS Filter	Signal Processing Blockset

See "Adaptive Filters" on page 3-53 for related information.

RMS

Purpose   Compute root-mean-square (RMS) value of an input or sequence of inputs

Library   Statistics
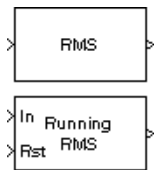
Description   The RMS block computes the RMS value of each column in the input, or tracks the RMS value of a sequence of inputs over a period of time. The **Running RMS** parameter selects between basic operation and running operation.

Basic Operation

When you do *not* select the **Running RMS** check box, the block computes the RMS value of each column in M-by-N input matrix u independently at each sample time.

```
y = sqrt(sum(u.*conj(u))/size(u,1))
```

For convenience, length-M 1-D vector inputs and *sample-based* length-M row vector inputs are both treated as M-by-1 column vectors.

The output at each sample time, y, is a 1-by-N vector containing the RMS value for each column in u. The RMS value of the *j*th column is

$$y_j = \sqrt{\frac{\sum_{i=1}^{M} u_{ij}^2}{M}}$$

The frame status of the output is the same as that of the input.

Running Operation

When you select the **Running RMS** check box, the block tracks the RMS value of each channel in a *time-sequence* of M-by-N inputs. For sample-based inputs, the output is a sample-based M-by-N matrix with each element y_{ij} containing the RMS value of element u_{ij} over all inputs since the last reset. For frame-based inputs, the output is a frame-based M-by-N matrix with each element y_{ij} containing the RMS

value of the *j*th column over all inputs since the last reset, up to and including element u_{ij} of the current input.

As in basic operation, length-M 1-D vector inputs and *sample-based* length-M row vector inputs are both treated as M-by-1 column vectors.

Resetting the Running RMS

The block resets the running RMS whenever a reset event is detected at the optional Rst port. The reset signal rate must be a positive integer multiple of the rate of the data signal input.

When the block is reset for sample-based inputs, the running RMS for each channel is initialized to the value in the corresponding channel of the current input. For frame-based inputs, the running RMS for each channel is initialized to the earliest value in each channel of the current input.

You specify the reset event in the **Reset port** parameter:

- None disables the Rst port.

- Rising edge – Triggers a reset operation when the Rst input does one of the following:

  - Rises from a negative value to a positive value or zero

  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero; see the following figure

- `Falling edge` – Triggers a reset operation when the `Rst` input does one of the following:

  - Falls from a positive value to a negative value or zero

  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero; see the following figure

- `Either edge` – Triggers a reset operation when the `Rst` input is a `Rising edge` or `Falling edge`, as described above.

- `Non-zero sample` – Triggers a reset operation at each sample time that the `Rst` input is not zero.

Note When running simulations in the Simulink `MultiTasking` mode, reset signals have a one-sample latency. Therefore, when the block detects a reset event, there is a one-sample delay at the reset port rate before the block applies the reset. For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" on page 2-56 and "Models with Multiple Sample Rates" in the Real-Time Workshop User's Guide documentation.

Examples   The RMS block in the model below calculates the running RMS of a frame-based 3-by-2 (two-channel) matrix input, u. The running RMS is reset at *t*=2 by an impulse to the block's Rst port.
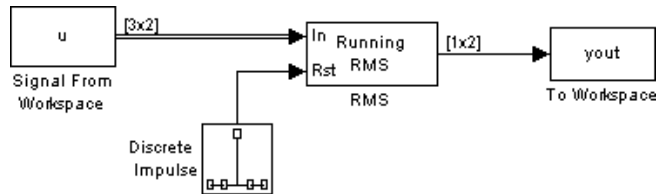
The RMS block has the following settings:

- **Running RMS** = Select this check box
- **Reset port** = Non-zero sample

The Signal From Workspace block has the following settings:

- **Signal** = u
- **Sample time** = 1/3
- **Samples per frame** = 3

where

```
u = [6 1 3 -7 2 5 8 0 -1 -3 2 1;1 3 9 2 4 1 6 2 5 0 4 17]'
```

The Discrete Impulse block has the following settings:

- **Delay (samples)** = 2
- **Sample time** = 1
- **Samples per frame** = 1

The block's operation is shown in the figure below.

RMS
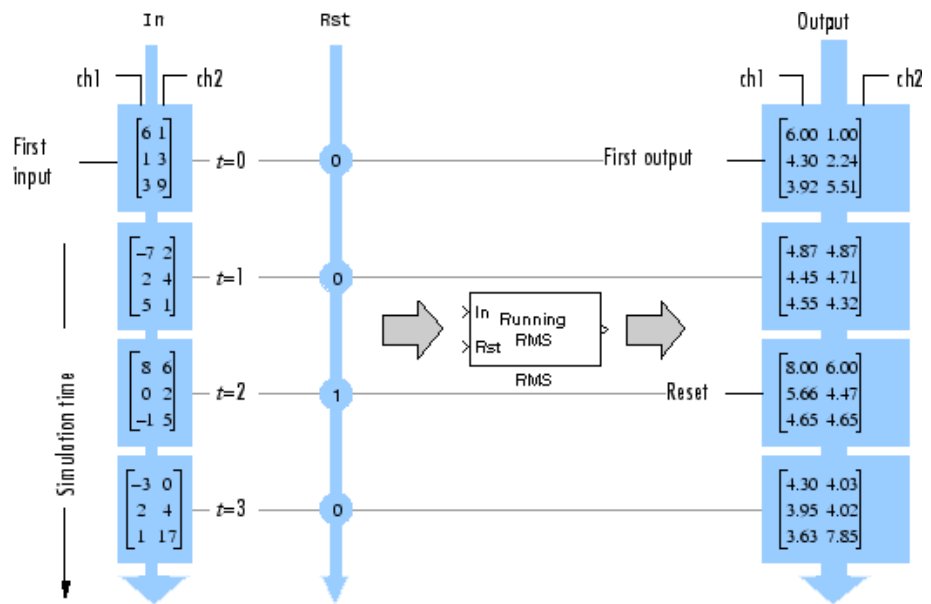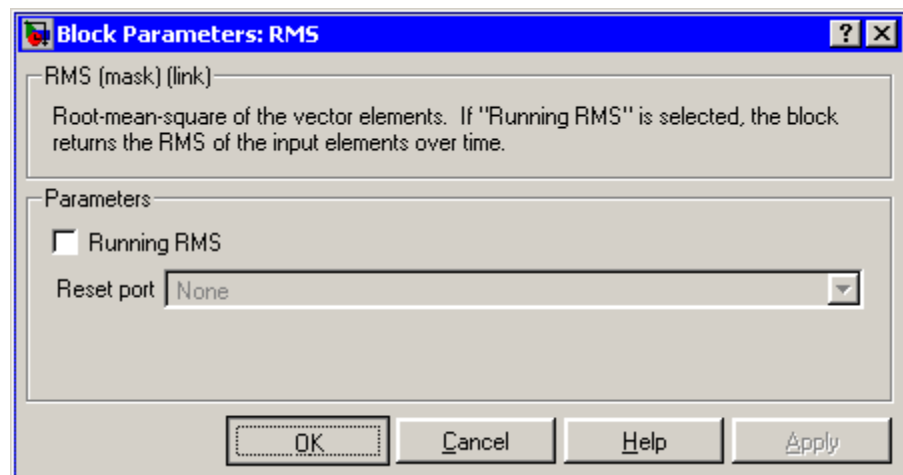
Running RMS

Enables running operation when selected.

Reset port

Determines the reset event that causes the block to reset the running RMS. The reset signal rate must be a positive integer multiple of the rate of the data signal input. This parameter is enabled only when you set the **Running RMS** parameter. For more information, see "Resetting the Running RMS" on page 10-899.

Supported Data Types

- Double-precision floating point

- Single-precision floating point

- Boolean – The block accepts Boolean inputs to the Rst port.

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Mean	Signal Processing Blockset
Variance	Signal Processing Blockset

Sample and Hold

Purpose   Sample and hold input signal

Library   Signal Operations
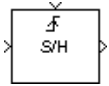
Description   The Sample and Hold block acquires the input at the signal port whenever it receives a trigger event at the trigger port (marked by ⨍). The block then holds the output at the acquired input value until the next triggering event occurs. When the acquired input is frame based, the output is frame based; otherwise, the output is sample based.

The trigger input must be a sample-based scalar with sample rate equal to the input frame rate at the signal port. You specify the trigger event in the **Trigger type** pop-up menu:

- `Rising edge` triggers the block to acquire the signal input when the trigger input rises from a negative value or zero to a positive value.

- `Falling edge` triggers the block to acquire the signal input when the trigger input falls from a positive value or zero to a negative value.

- `Either edge` triggers the block to acquire the signal input when the trigger input either rises from a negative value or zero to a positive value or falls from a positive value or zero to a negative value.

You specify the block's output prior to the first trigger event using the **Initial condition** parameter. When the acquired input is an M-by-N matrix, the **Initial condition** can be an M-by-N matrix, or a scalar to be repeated across all elements of the matrix. When the input is a length-M 1-D vector, the **Initial condition** can be a length-M row or column vector, or a scalar to be repeated across all elements of the vector.

If you select the **Latch (buffer) input** check box, the block outputs the value of the input from the previous time step until the next triggering event occurs. To use this block in a loop, select this check box.

**Dialog
Box**
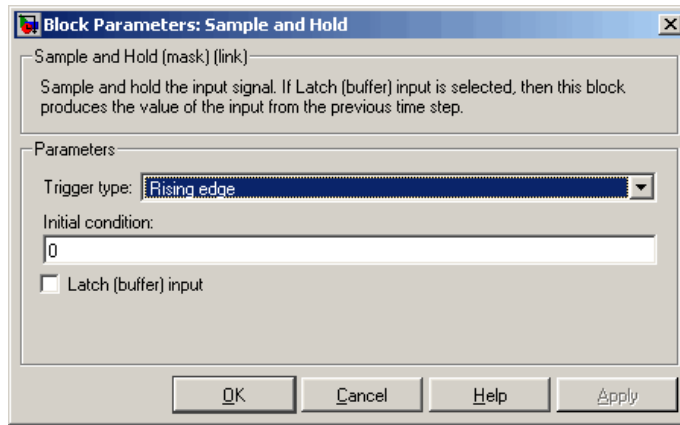
Trigger type

> The type of event that triggers the block to acquire the input signal.

Initial condition

> The block's output prior to the first trigger event.

Latch (buffer) input

> If you select this check box, the block outputs the value of the input from the previous time step until the next triggering event occurs.

Sample and Hold

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Custom data types
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
Trigger	• Any data type supported by the Trigger block
Outputs	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Custom data types
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Downsample	Signal Processing Blockset
N-Sample Switch	Signal Processing Blockset

Purpose   Convert an input signal into a set of quantized output values. Convert an input signal into a set of index values. Convert a set of index values into a quantized output signal.

Library   dspobslib

Description
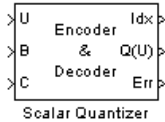
Scalar Quantizer

Note The Scalar Quantizer block is still supported but is likely to be obsoleted in a future release. We recommend replacing this block with the Scalar Quantizer Encoder block or the Scalar Quantizer Decoder block.

The Scalar Quantizer block has three modes of operation. In Encoder mode, the block maps each input value to a quantization region by comparing the input value to the quantizer boundary points defined in the **Boundary points** parameter. The block outputs the index of the associated region. In Decoder mode, the block transforms the input index values into quantized output values, defined in the **Codebook** parameter. In the Encoder and Decoder mode, the block performs both the encoding and decoding operations. The block outputs the index values and the quantized output values.

You can select how you want to enter the **Boundary points** and/or **Codebook** values using the **Source of quantizer** parameters. When you select Specify via dialog, type the parameters into the block parameters dialog box. Select Input ports, and port B and/or C appears on the block. In Encoder and Encoder and decoder mode, the input to port B is used as the **Boundary points**. In Decoder and Encoder and decoder mode, the input to port C is used as the **Codebook**.

In Encoder and Encoder and decoder mode, the **Boundary points** are the values used to break up the input signal into regions. Each region is specified by an index number. When your first boundary point is -inf and your last boundary point is inf, your quantizer is unbounded. When your first and last boundary point is finite, your

Scalar Quantizer

quantizer is bounded. When only your first or last boundary point is -inf or inf, your quantizer is semi-bounded.

For instance, when your input signal ranges from 0 to 11, you can create a bounded quantizer using the following boundary points:

```
[0 0.5 3.7 5.8 6.0 11]
```

The boundary points can have equal or varied spacing. Any input values between 0 and 0.5 would correspond to index 0. Input values between 0.5 and 3.7 would correspond to index 1, and so on.

Suppose you wanted to create an unbounded quantizer with the following boundary points:

```
[-inf 0 2 5.5 7.1 10 inf]
```

When your input signal has values less than 0, these values would be assigned to index 0. When your input signal has values greater than 10, these values would be assigned to index 6.

When an input value is the same as a boundary point, the **Tie-breaking rule** parameter defines the index to which the value is assigned. When you want the input value to be assigned to the lower index value, select Choose the lower index. To assign the input value with the higher index, select Choose the higher index.

In Decoder and Encoder and decoder mode, the **Codebook** is a vector of quantized output values that correspond to each index value.

In Encoder and Encoder and decoder mode, the **Searching method** determines how the appropriate quantizer index is found. Select Linear and the Scalar Quantizer block compares the input value to the first region defined by the first two boundary points. When the input value does not fall within this region, the block then compares the input value to the next region. This process continues until the input value is determined to be within a region and is associated with the appropriate index value. The computational cost of this process is of the order P, where P is the number of boundary points.

Select `Binary` for the **Searching method** and the block compares the input value to the middle value of the boundary points vector. When the input value is larger than this boundary point, the block discards the boundary points that are lower than this middle value. The block then compares the input value to the middle boundary point of the new range, defined by the remaining boundary points. This process continues until the input value is associated with the appropriate index value. The computational cost of this process is of the order $\log_2 P$, where P is the number of boundary points. In most cases, the `Binary` option is faster than the `Linear` option.

In `Decoder` mode, the input to this block is a vector of index values, where $0 \leq \text{index} < N$ and N is the length of the codebook vector. Use the **Action for out of range input** parameter to determine what happens when an input index value is out of this range. When you want any index values less than 0 to be set to 0 and any index values greater than or equal to N to be set to N-1, select `Clip`. When you want to be warned when any index values less than 0 are set to 0 and any index values greater than or equal to N are set to N-1, select `Clip and warn`. When you want the simulation to stop and display an error when the index values are out of range, select `Error`.

In `Encoder and decoder` mode, you can select the **Output the quantization error** check box. The quantization error is the difference between the input value and the quantized output value. Select this check box to output the quantization error for each input value from the Err port on this block.

Data Type Support

In `Encoder` mode, the input data values and the boundary points can be the input to the block at ports U and B. Similarly, in `Encoder and decoder` mode, the codebook values can also be the input to the block at port C. The data type of the input data values, boundary points, and codebook values can be `double`, `single`, `uint8`, `uint16`, `uint32`, `int8`, `int16`, or `int32`. In `Decoder` mode, the input to the block can be the index values and the codebook values. The data type of the index input to the block at port Idx can be `uint8`, `uint16`, `uint32`, `int8`, `int16`, or
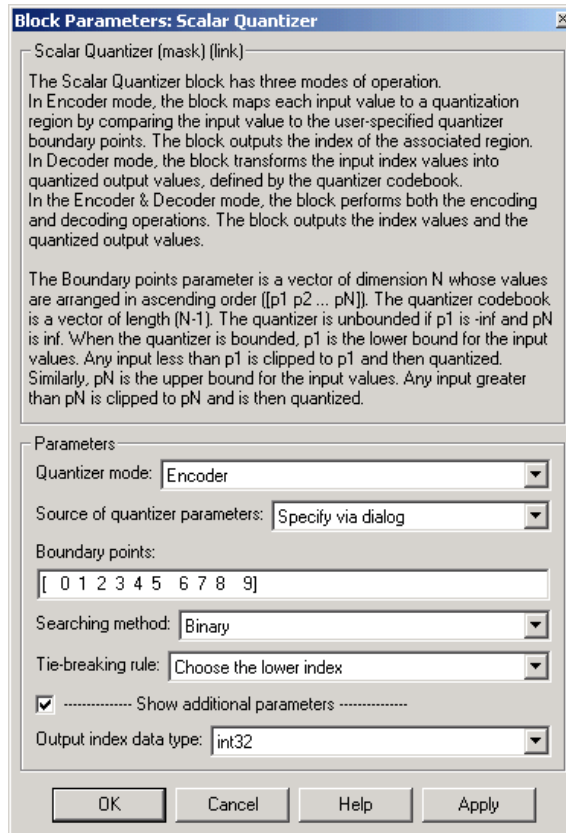
int32. The data type of the codebook values can be `double`, `single`, `uint8`, `uint16`, `uint32`, `int8`, `int16`, or `int32`.

In `Encoder` mode, the output of the block is the index values. In `Encoder and decoder` mode, the output can also include the quantized output values and the quantization error. In `Encoder` and `Encoder and decoder` mode, use the **Output index data type** parameter to specify the data type of the index output from the block at port Idx. The data type of the index output can be `uint8`, `uint16`, `uint32`, `int8`, `int16`, or `int32`. The data type of the quantized output and the quantization error can be `double`, `single`, `uint8`, `uint16`, `uint32`, `int8`, `int16`, or `int32`. In `Decoder` mode, the output of the block is the quantized output values. Use the **Output data type** parameter to specify the data type of the quantized output values. The data type can be `double`, `single`, `uint8`, `uint16`, `uint32`, `int8`, `int16`, `int32`.

Note The input data, codebook values, boundary points, quantization error, and the quantized output values must have the same data type whenever they are present in any of the quantizer modes.

**Dialog
Box**

Block Parameters: Scalar Quantizer

Scalar Quantizer (mask) (link)

The Scalar Quantizer block has three modes of operation.
In Encoder mode, the block maps each input value to a quantization region by comparing the input value to the user-specified quantizer boundary points. The block outputs the index of the associated region. In Decoder mode, the block transforms the input index values into quantized output values, defined by the quantizer codebook. In the Encoder & Decoder mode, the block performs both the encoding and decoding operations. The block outputs the index values and the quantized output values.

The Boundary points parameter is a vector of dimension N whose values are arranged in ascending order ([p1 p2 ... pN]). The quantizer codebook is a vector of length (N-1). The quantizer is unbounded if p1 is -inf and pN is inf. When the quantizer is bounded, p1 is the lower bound for the input values. Any input less than p1 is clipped to p1 and then quantized. Similarly, pN is the upper bound for the input values. Any input greater than pN is clipped to pN and is then quantized.

Parameters

Quantizer mode: Encoder

Source of quantizer parameters: Specify via dialog

Boundary points:

[0 1 2 3 4 5 6 7 8 9]

Searching method: Binary

Tie-breaking rule: Choose the lower index

☑ -------------- Show additional parameters --------------

Output index data type: int32

| OK | Cancel | Help | Apply |

Scalar Quantizer

Block Parameters: Scalar Quantizer

Scalar Quantizer (mask) (link)

The Scalar Quantizer block has three modes of operation.
In Encoder mode, the block maps each input value to a quantization region by comparing the input value to the user-specified quantizer boundary points. The block outputs the index of the associated region.
In Decoder mode, the block transforms the input index values into quantized output values, defined by the quantizer codebook.
In the Encoder & Decoder mode, the block performs both the encoding and decoding operations. The block outputs the index values and the quantized output values.

The Boundary points parameter is a vector of dimension N whose values are arranged in ascending order ([p1 p2 ... pN]). The quantizer codebook is a vector of length (N-1). The quantizer is unbounded if p1 is -inf and pN is inf. When the quantizer is bounded, p1 is the lower bound for the input values. Any input less than p1 is clipped to p1 and then quantized. Similarly, pN is the upper bound for the input values. Any input greater than pN is clipped to pN and is then quantized.

Parameters

Quantizer mode: Decoder

Source of quantizer parameters: Specify via dialog

Codebook:

[0.0   1.5   2.5   3.5   4.5   5.5   6.5 7.5 8.5]

Action for out of range input: Clip and warn

☑ -------------- Show additional parameters --------------

Output data type: double

OK   Cancel   Help   Apply

Quantizer mode

Specify Encoder, Decoder, or Encoder and decoder as a mode of operation. Nontunable.

Source of quantizer parameters

Choose Specify via dialog to type the parameters into the block parameters dialog box. Select Input ports to specify the

Scalar Quantizer

parameters using the block's input ports. In `Encoder` and `Encoder and decoder` mode, input the **Boundary points** using port B. In `Decoder` and `Encoder and decoder` mode, input the **Codebook** values using port C. Nontunable.

Boundary points
Enter a vector of values that represent the boundary points of the quantizer regions. Tunable.

Codebook
Enter a vector of quantized output values that correspond to each index value. Tunable.

Searching method
Select `Linear` and the block finds the region in which the input value is located using a linear search. Select `Binary` and the block finds the region in which the input value is located using a binary search. Nontunable.

Tie-breaking rule
Set this parameter to determine the behavior of the block when the input value is the same as the boundary point. When you select `Choose the lower index`, the input value is assigned to lower index value. When you select `Choose the higher index`, the value is assigned to the higher index. Nontunable.

Action for out of range input
Choose the block's behavior when an input index value is out of range, where $0 \le \text{index} < N$ and N is the length of the codebook vector. Select `Clip`, when you want any index values less than 0 to be set to 0 and any index values greater than or equal to N to be set to N-1. Select `Clip and warn`, when you want to be warned when any index values less than 0 are set to 0 and any index values greater than or equal to N are set to N-1. Select `Error`, when you want the simulation to stop and display an error when the index values are out of range. Nontunable.

Output the quantization error
In `Encoder and decoder` mode, select this check box to output the quantization error from the Err port on this block. Nontunable.

Output index data type

In `Encoder` and `Encoder` and `decoder` mode, specify the data type of the index output from the block at port Idx. The data type can be `uint8`, `uint16`, `uint32`, `int8`, `int16`, or `int32`. This parameter becomes visible when you select the **Show additional parameters** check box. Nontunable.

Output data type

In `Decoder` mode, specify the data type of the quantized output. The data type can be `uint8`, `uint16`, `uint32`, `int8`, `int16`, `int32`, `single`, or `double`. This parameter becomes visible when you select `Specify via dialog` for the **Source of quantizer parameters** and you select the **Show additional parameters** check box. Nontunable.

References   Gersho, A. and R. Gray. *Vector Quantization and Signal Compression*. Boston: Kluwer Academic Publishers, 1992.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

For more information on what data types are supported for each quantizer mode, see "Data Type Support" on page 10-909. To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Quantizer	Simulink
Scalar Quantizer Decoder	Signal Processing Blockset

Scalar Quantizer

Scalar Quantizer Design	Signal Processing Blockset
Scalar Quantizer Encoder	Signal Processing Blockset
Uniform Encoder	Signal Processing Blockset
Uniform Decoder	Signal Processing Blockset

Purpose   Convert each index value into quantized output value

Library   Quantizers
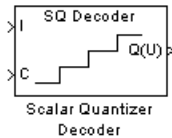
Description   The Scalar Quantizer Decoder block transforms the zero-based input index values into quantized output values. The set of all possible quantized output values is defined by the **Codebook values** parameter.

Use the **Codebook values** parameter to specify a matrix containing all possible quantized output values. You can select how you want to enter the codebook values using the **Source of codebook** parameter. When you select Specify via dialog, type the codebook values into the block parameters dialog box. When you select Input port, port C appears on the block. The block uses the input to port C as the **Codebook values** parameter.

The input to this block is a vector of integer index values, where $0 \le index < N$ and N is the number of distinct codeword vectors in the codebook matrix. Use the **Action for out of range index value** parameter to determine what happens when an input index value is outside this range. When you want any index value less than 0 to be set to 0 and any index value greater than or equal to N to be set to N-1, select Clip. When you want to be warned when clipping occurs, select Clip and warn. When you want the simulation to stop and the block to display an error when the index values are out of range, select Error.

Data Type Support

The data type of the index values input at port I can be uint8, uint16, uint32, int8, int16, or int32. The data type of the codebook values input at port C can be double, single, or Fixed-point.

The output of the block is the quantized output values. If, for the **Source of codebook** parameter, you select Specify via dialog, the **Codebook and output data type** parameter appears. You can use this parameter to specify the data type of the codebook and quantized output values. In this case, the data type of the output values can be Same as input, double, single, Fixed-point, or User-defined. If, for the **Source of codebook** parameter you select Input port,

Scalar Quantizer Decoder

the quantized output values have the same data type as the codebook values input at port C.

Dialog Box
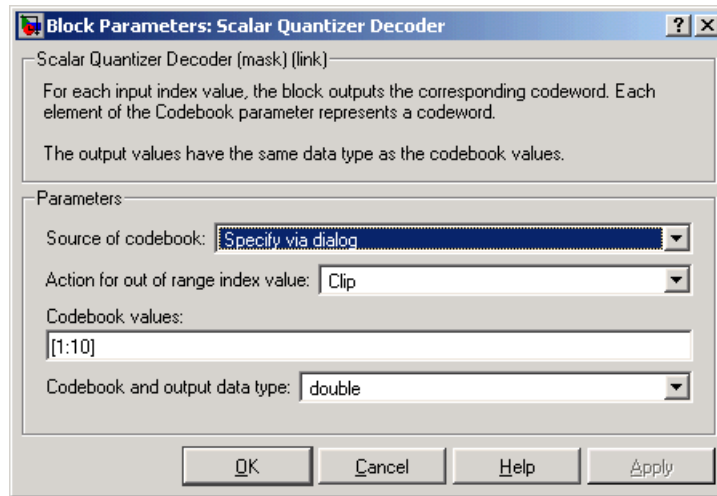
Source of codebook
Choose `Specify via dialog` to type the codebook values into the block parameters dialog box. Select `Input port` to specify the codebook using input port C. Nontunable.

Action for out of range index value
Use this parameter to determine the block's behavior when an input index value is out of range, where $0 \leq index < N$ and N is the length of the codebook vector. Select `Clip`, when you want any index values less than 0 to be set to 0 and any index values greater than or equal to N to be set to N-1. Select `Clip and warn`, when you want to be warned when clipping occurs. Select `Error`, when you want the simulation to stop and the block to display an error when the index values are outside the range. Nontunable.

Codebook values
Enter a vector of quantized output values that correspond to each index value. Tunable.

Codebook and output data type

Use this parameter to specify the data type of the codebook and quantized output values. The data type can be Same as input, double, single, Fixed-point, or User-defined. This parameter becomes visible when you select Specify via dialog for the **Source of codebook** parameter. Nontunable.
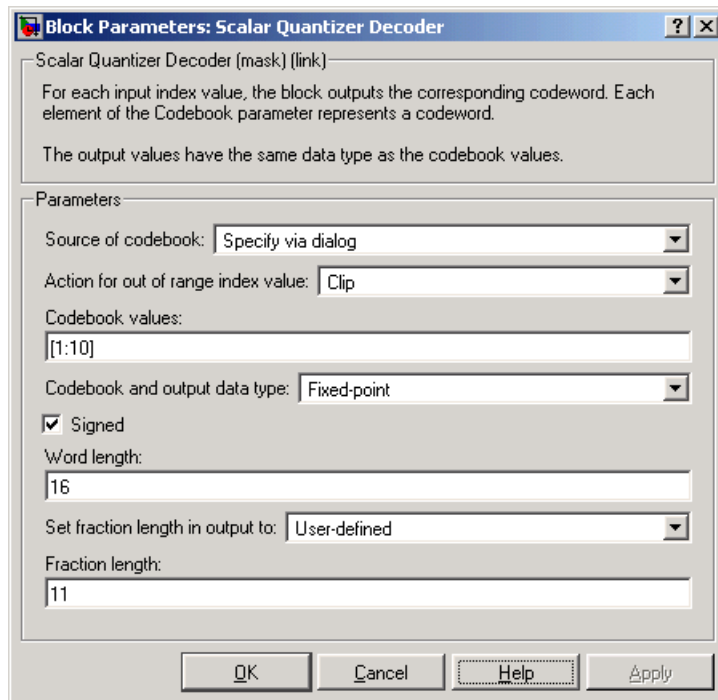
Signed

Select to output a signed fixed-point signal. Otherwise, the signal will be unsigned. This parameter is only visible if, from the **Codebook and output data type** list, you select Fixed-point.

Scalar Quantizer Decoder

Word length

Specify the word length, in bits, of the fixed-point output data type. This parameter is only visible if, from the **Codebook and output data type** list, you select Fixed-point.

Set fraction length in output to

Specify the scaling of the fixed-point output by either of the following two methods:

Choose Best precision to have the output scaling automatically set such that the output signal has the best possible precision.

Choose User-defined to specify the output scaling in the **Fraction length** parameter.

This parameter is only visible if, from the **Codebook and output data type** list, you select Fixed-point or when you select User-defined and the specified output data type is a fixed-point data type.

Fraction length

For fixed-point output data types, specify the number of fractional bits, or bits to the right of the binary point. This parameter is only visible when you select Fixed-point or User-defined for the **Codebook and output data type** parameter and User-defined for the **Set fraction length in output to** parameter.

User-defined data type

Specify any built-in or fixed-point data type. You can specify fixed-point data types using the sfix, ufix, sint, uint, sfrac, and ufrac functions from Simulink Fixed Point. This parameter is only visible when you select User-defined for the **Codebook and output data type** parameter.

References   Gersho, A. and R. Gray. *Vector Quantization and Signal Compression*. Boston: Kluwer Academic Publishers, 1992.

Scalar Quantizer Decoder

Supported Data Types

Port	Supported Data Types
I	• 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
C	• Double-precision floating point • Single-precision floating point • Fixed point • 8-, 16-, and 32-bit signed integers
Q(U)	• Double-precision floating point • Single-precision floating point • Fixed point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

For more information on what data types are supported for each quantizer mode, see "Data Type Support" on page 10-917. To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Quantizer	Simulink
Scalar Quantizer Design	Signal Processing Blockset
Scalar Quantizer Encoder	Signal Processing Blockset
Uniform Encoder	Signal Processing Blockset
Uniform Decoder	Signal Processing Blockset

Purpose Start Scalar Quantizer Design Tool (SQDTool) to design scalar quantizer using Lloyd algorithm

Library Quantizers
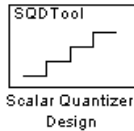
Description Double-click on the Scalar Quantizer Design block to start SQDTool, a GUI that allows you to design and implement a scalar quantizer. Based on your input values, SQDTool iteratively calculates the codebook values that minimize the mean squared error until the stopping criteria for the design process is satisfied. The block uses the resulting quantizer codebook values and boundary points to implement your scalar quantizer encoder and/or decoder.

For the **Training Set** parameter, enter a set of observations, or samples, of the signal you want to quantize. This data can be any variable defined in the MATLAB workspace including a variable created using a MATLAB function, such as the default value randn(10000,1).

You have two choices for the **Source of initial codebook** parameter. Select Auto-generate to have the block choose the values of the initial codebook vector. In this case, the minimum training set value becomes the first codeword, and the maximum training set value becomes the last codeword. Then, the remaining initial codewords are equally spaced between these two values to form a codebook vector of length N, where N is the **Number of levels** parameter. When you select User defined, enter the initial codebook values in the **Initial codebook** field. Then, set the **Source of initial boundary points** parameter. You can select Mid-points to locate the boundary points at the midpoint between the codewords. To calculate the mid-points, the block internally arranges the initial codebook values in ascending order. You can also choose User defined and enter your own boundary points in the **Initial boundary points (unbounded)** field. Only one boundary point can be located between two codewords. When you select User defined for the **Source of initial boundary points** parameter, the values you enter in the **Initial codebook** and **Initial boundary points (unbounded)** fields must be arranged in ascending order.

Scalar Quantizer Design

Note This block assumes that you are designing an unbounded quantizer. Therefore, the first and last boundary points are always `-inf` and `inf` regardless of any other boundary point values you might enter.

After you have specified the quantization parameters, the block performs an iterative process to design the optimal scalar quantizer. Each step of the design process involves using the Lloyd algorithm to calculate codebook values and quantizer boundary points. Then, the block calculates the squared quantization error and checks whether the stopping criteria has been satisfied.

The two possible options for the **Stopping criteria** parameter are `Relative threshold` and `Maximum iteration`. When you want the design process to stop when the fractional drop in the squared quantization error is below a certain value, select `Relative threshold`. Then, for **Relative threshold**, type the maximum acceptable fractional drop. When you want the design process to stop after a certain number of iterations, choose `Maximum iteration`. Then, enter the maximum number of iterations you want the block to perform in the **Maximum iteration** field. For **Stopping criteria**, you can also choose `Whichever comes first` and enter a **Relative threshold** and **Maximum iteration** value. The block stops iterating as soon as one of these conditions is satisfied.

With each iteration, the block quantizes the training set values based on the newly calculated codebook values and boundary points. When the training point lies on a boundary point, the algorithm uses the **Tie-breaking rules** parameter to determine which region the value is associated with. When you want the training point to be assigned to the lower indexed region, select `Lower indexed codeword`. To assign the training point with the higher indexed region, select `Higher indexed codeword`.

The **Searching methods** parameter determines how the block compares the training points to the boundary points. Select `Linear search` and SQDTool compares each training point to each quantization

region sequentially. This process continues until all the training points are associated with the appropriate regions.

Select Binary search for the **Searching methods** parameter and the block compares the training point to the middle value of the boundary points vector. When the training point is larger than this boundary point, the block discards the lower boundary points. The block then compares the training point to the middle boundary point of the new range, defined by the remaining boundary points. This process continues until the training point is associated with the appropriate region.

Click **Design and Plot** to design the quantizer with the parameter values specified on the left side of the GUI. The performance curve and the staircase character of the quantizer are updated and displayed in the figures on the right side of the GUI.

Note You must click **Design and Plot** to apply any changes you make to the parameter values in the SQDTool dialog box.

SQDTool can export parameter values that correspond to the figures displayed in the GUI. Click the **Export Outputs** button, or press **Ctrl+E**, to export the **Final Codebook**, **Final Boundary Points**, and **Error** values to the workspace, a text file, or a MAT-file. The **Error** values represent the mean squared error for each iteration.

In the **Model** section of the GUI, specify the destination of the block that will contain the parameters of your quantizer. For **Destination**, select Current model to create a block with your parameters in the model you most recently selected. Type gcs in the MATLAB Command Window to display the name of your current model. Select New model to create a block in a new model file.

From the **Block type** list, select Encoder to design a Scalar Quantizer Encoder block. Select Decoder to design a Scalar Quantizer Decoder block. Select Both to design a Scalar Quantizer Encoder block and a Scalar Quantizer Decoder block.

In the **Encoder block name** field, enter a name for the Scalar Quantizer Encoder block. In the **Decoder block name** field, enter a name for the Scalar Quantizer Decoder block. When you have a Scalar Quantizer Encoder and/or Decoder block in your destination model with the same name, select the **Overwrite target block(s)** check box to replace the block's parameters with the current parameters. When you do not select this check box, a new Scalar Quantizer Encoder and/or Decoder block is created in your destination model.

Click **Generate Model**. SQDTool uses the parameters that correspond to the current plots to set the parameters of the Scalar Quantizer Encoder and/or Decoder blocks.

Dialog Box

Scalar Quantizer Design

Training Set

Enter the samples of the signal you would like to quantize. This data set can be a MATLAB function or a variable defined in the MATLAB workspace. The typical length of this data vector is `1e6`.

Source of initial codebook

Select `Auto-generate` to have the block choose the initial codebook values. Select `User defined` to enter your own initial codebook values.

Number of levels

Enter the length of the codebook vector. For a b-bit quantizer, the length should be $N = 2^b$.

Initial codebook

Enter your initial codebook values. From the **Source of initial codebook** list, select `User defined` in order to activate this parameter.

Source of initial boundary points

Select `Mid-points` to locate the boundary points at the midpoint between the codebook values. Choose `User defined` to enter your own boundary points. From the **Source of initial codebook** list, select `User defined` in order to activate this parameter.

Initial boundary points (unbounded)

Enter your initial boundary points. This block assumes that you are designing an unbounded quantizer. Therefore, the first and last boundary point are `-inf` and `inf`, regardless of any other boundary point values you might enter. From the **Source of initial boundary points** list, select `User defined` in order to activate this parameter.

Stopping criteria

Choose `Relative threshold` to enter the maximum acceptable fractional drop in the squared quantization error. Choose `Maximum iteration` to specify the number of iterations at which to stop. Choose `Whichever comes first` and the block stops the iteration process as soon as the relative threshold or maximum iteration value is attained.

Relative threshold

Type the value that is the maximum acceptable fractional drop in the squared quantization error.

Maximum iteration

Enter the maximum number of iterations you want the block to perform. From the **Stopping criteria** list, select `Maximum iteration` in order to activate this parameter.

Searching methods

Choose `Linear search` to use a linear search method when comparing the training points to the boundary points. Choose `Binary search` to use a binary search method when comparing the training points to the boundary points.

Tie-breaking rules

When a training point lies on a boundary point, choose `Lower indexed codeword` to assign the training point to the lower indexed quantization region. Choose `Higher indexed codeword` to assign the training point to the higher indexed region.

Design and Plot

Click this button to display the performance curve and the staircase character of the quantizer in the figures on the right side of the GUI. These plots are based on the current parameter settings.

You must click **Design and Plot** to apply any changes you make to the parameter values in the SQDTool GUI.

Export Outputs

Click this button, or press **Ctrl+E**, to export the **Final Codebook**, **Final Boundary Points**, and **Error** values to the workspace, a text file, or a MAT-file.

Destination

Choose `Current model` to create a Scalar Quantizer block in the model you most recently selected. Type `gcs` in the MATLAB Command Window to display the name of your current model. Choose `New model` to create a block in a new model file.

Block type

Select Encoder to design a Scalar Quantizer Encoder block. Select Decoder to design a Scalar Quantizer Decoder block. Select Both to design a Scalar Quantizer Encoder block and a Scalar Quantizer Decoder block.

Encoder block name

Enter a name for the Scalar Quantizer Encoder block.

Decoder block name

Enter a name for the Scalar Quantizer Decoder block.

Overwrite target block(s)

When you do not select this check box and a Scalar Quantizer Encoder and/or Decoder block with the same block name exists in the destination model, a new Scalar Quantizer Encoder and/or Decoder block is created in the destination model. When you select this check box and a Scalar Quantizer Encoder and/or Decoder block with the same block name exists in the destination model, the parameters of these blocks are overwritten by new parameters.

Generate Model

Click this button and SQDTool uses the parameters that correspond to the current plots to set the parameters of the Scalar Quantizer Encoder and/or Decoder blocks.

References Gersho, A. and R. Gray. *Vector Quantization and Signal Compression*. Boston: Kluwer Academic Publishers, 1992.

Supported Data Types

- Double-precision floating point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Quantizer	Simulink
Scalar Quantizer Decoder	Signal Processing Blockset
Scalar Quantizer Encoder	Signal Processing Blockset
Uniform Encoder	Signal Processing Blockset
Uniform Decoder	Signal Processing Blockset

Scalar Quantizer Encoder

Purpose
Encode each input value by associating it with the index value of a quantization region

Library
Quantizers

Description

Scalar Quantizer Encoder
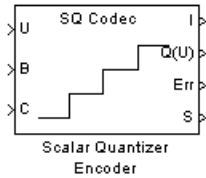
The Scalar Quantizer Encoder block maps each input value to a quantization region by comparing the input value to the quantizer boundary points defined in the **Boundary points** parameter. The block outputs the zero-based index of the associated region.

You can select how you want to enter the **Boundary points** using the **Source of quantizer parameters**. When you select Specify via dialog, type the boundary points into the block parameters dialog box. When you select Input port, port B appears on the block. The block uses the input to port B as the **Boundary points** parameter.

Use the **Boundary points** parameter to specify the boundary points for your quantizer. These values are used to break up the set of input numbers into regions. Each region is specified by an index number.

Let N be the number of quantization regions. When the codebook is defined as [c1 c2 c3 ... cN], and the **Boundary points** parameter is defined as [p0 p1 p2 p3 ... pN], then p0<c1<p1<c2 ... p(N-1)<cN<pN for a regular quantizer. When your quantizer is bounded, from the **Partitioning** list, select Bounded. You need to specify N+1 boundary points, or [p0 p1 p2 p3 ... pN]. When your quantizer is unbounded, from the **Partitioning** list, select Unbounded. You need to specify N-1 boundary points, or [p1 p2 p3 ... p(N-1)]; the block sets p0 equal to -inf and pN equal to inf.

The block uses the **Partitioning** parameter to interpret the boundary points you enter. For instance, to create a bounded quantizer, from the **Partitioning** list, select Bounded and enter the following boundary points:

    [0 0.5 3.7 5.8 6.0 11]

The block assigns any input values between 0 and 0.5 to index 0, input values between 0.5 and 3.7 to index 1, and so on. The block assigns

any values that are less than 0 to index 0, the lowest index value. The block assigns any values that are greater than 11 to index 4, the highest index value.

To create an unbounded quantizer, from the **Partitioning** list, select Unbounded and enter the following boundary points:

```
[0 0.5 3.7 5.8 6.0 11]
```

The block assigns any input values between 0 and 0.5 to index 1, input values between 0.5 and 3.7 to index 2, and so on. The block assigns any input values less than 0 to index 0 and any values greater than 11 to index 6.

The **Searching method** parameter determines how the appropriate quantizer index is found. When you select Linear, the Scalar Quantizer Encoder block compares the input value to the first region defined by the first two boundary points. When the input value does not fall within this region, the block then compares the input value to the next region. This process continues until the input value is determined to be within a region and is associated with the appropriate index value. The computational cost of this process is of the order P, where P is the number of boundary points.

When you select Binary for the **Searching method**, the block compares the input value to the middle value of the boundary points vector. When the input value is larger than this boundary point, the block discards the boundary points that are lower than this middle value. The block then compares the input value to the middle boundary point of the new range, defined by the remaining boundary points. This process continues until the input value is associated with the appropriate index value. The computational cost of this process is of the order $\log_2 P$, where P is the number of boundary points. In most cases, the Binary option is faster than the Linear option.

When an input value is the same as a boundary point, the **Tie-breaking rule** parameter determines the region to which the value is assigned. When you want the input value to be assigned to the lower indexed

region, select `Choose the lower index`. To assign the input value with the higher indexed region, select `Choose the higher index`.

Select the **Output codeword** check box to output the codeword values that correspond to each index value at port Q(U).

Select the **Output the quantization error** check box to output the quantization error for each input value from the Err port on this block. The quantization error is the difference between the input value and the quantized output value.

When you select either the **Output codeword** check box or the **Output quantization error** check box, you must also enter your codebook values. If, from the **Source of quantizer parameters** list, you choose `Specify via dialog`, use the **Codebook** parameter to enter a vector of quantized output values that correspond to each region. If, from the **Source of quantizer parameters** list, you choose `Input port`, use input port C to specify your codebook values.

If, for the **Partitioning** parameter, you select `Bounded`, the **Output clipping status** check box and the **Action for out of range input** parameter appear. When you select the **Output clipping status** check box, port S appears on the block. Any time an input value is outside the range defined by the **Boundary points** parameter, the block outputs a 1 at the S port. When the value is inside the range, the blocks outputs a 0.

You can use the **Action for out of range input** parameter to determine the block's behavior when an input value is outside the range defined by the **Boundary points** parameter. Suppose the boundary points for a bounded quantizer are defined as `[p0 p1 p2 p3 ... pN]` and the possible index values are defined as `[i0 i1 i2 ... i(N-1)]`, where i0=0 and i0<i1<i2<...<i(N-1). When you want any input value less than p0 to be assigned to index value i0 and any input values greater than pN to be assigned to index value i(N-1), select `Clip`. When you want to be warned when clipping occurs, select `Clip and warn`. When you want the simulation to stop and the block to display an error when the index values are out of range, select `Error`.

The Scalar Quantizer Encoder block accepts real floating-point and fixed-point inputs. For more information on the data types accepted by each port, see "Data Type Support" on page 10-935 or "Supported Data Types" on page 10-940.

Data Type Support

The input data values, boundary points, and codebook values can be input to the block at ports U, B, and C, respectively. The data type of the inputs can be double, single, or Fixed-point.

The outputs of the block can be the index values, the quantized output values, the quantization error, and the clipping status. Use the **Index output data type** parameter to specify the data type of the index output from the block at port I. You can choose int8, uint8, int16, uint16, int32, or uint32. The data type of the quantized output and the quantization error can be double, single, or Fixed-point. The clipping status values output at port S are Boolean values.

Note The input data, boundary points, codebook values, quantized output values, and the quantization error must have the same data type whenever they are present.

Scalar Quantizer Encoder

Dialog Box
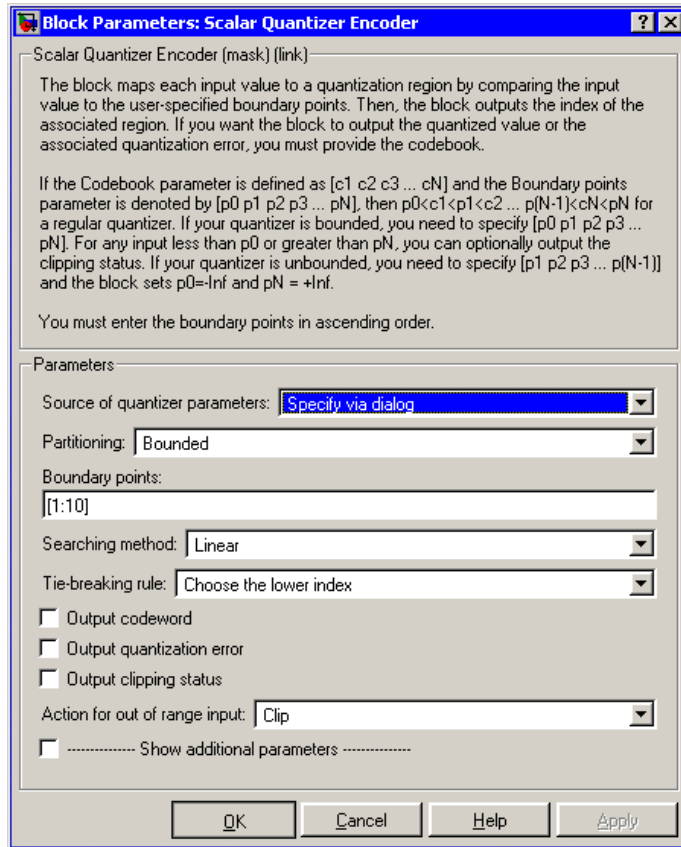
Source of quantizer parameters

Choose Specify via dialog to enter the boundary points and codebook values using the block parameters dialog box. Select Input port to specify the parameters using the block's input ports. Input the boundary points and codebook values using ports B and C, respectively. Nontunable.

Partitioning

When your quantizer is bounded, select Bounded. When your quantizer is unbounded, select Unbounded. Nontunable.

Boundary points

Enter a vector of values that represent the boundary points of the quantizer regions. This parameter is visible when you select Specify via dialog from the **Source of quantizer parameters** list. Tunable.

Searching method

When you select Linear, the block finds the region in which the input value is located using a linear search. When you select Binary, the block finds the region in which the input value is located using a binary search. Nontunable.

Tie-breaking rule

Set this parameter to determine the behavior of the block when the input value is the same as the boundary point. When you select Choose the lower index, the input value is assigned to lower indexed region. When you select Choose the higher index, the value is assigned to the higher indexed region. Nontunable.

Output codeword

Select this check box to output the codeword values that correspond to each index value at port Q(U). Nontunable.

Output quantization error

Select this check box to output the quantization error for each input value at port Err. Nontunable.

Codebook

Enter a vector of quantized output values that correspond to each index value. If, for the **Partitioning** parameter, you select Bounded and your boundary points vector has length N, then you must specify a codebook of length N-1. If, for the **Partitioning** parameter, you select Unbounded and your boundary points vector has length N, then you must specify a codebook of length N+1.

This parameter is visible when you select Specify via dialog from the **Source of quantizer parameters** list and you select either the **Output codeword** or **Output quantization error** check box. Tunable.

Output clipping status

When you select this check box, port S appears on the block. Any time an input value is outside the range defined by the **Boundary points** parameter, the block outputs a 1 at this port. When the value is inside the range, the block outputs a 0. This parameter is visible when you select `Bounded` from the **Partitioning** list.

Action for out of range input

Use this parameter to determine the behavior of the block when an input value is outside the range defined by the **Boundary points** parameter. Suppose the boundary points are defined as `[p0 p1 p2 p3 ...  pN]` and the index values are defined as `[i0 i1 i2 ...  i(N-1)]`. When you want any input value less than `p0` to be assigned to index value `i0` and any input values greater than `pN` to be assigned to index value `i(N-1)`, select `Clip`. When you want to be warned when clipping occurs, select `Clip and warn`. When you want the simulation to stop and the block to display an error when the index values are out of range, select `Error`. This parameter is visible when you select `Bounded` from the **Partitioning** list.
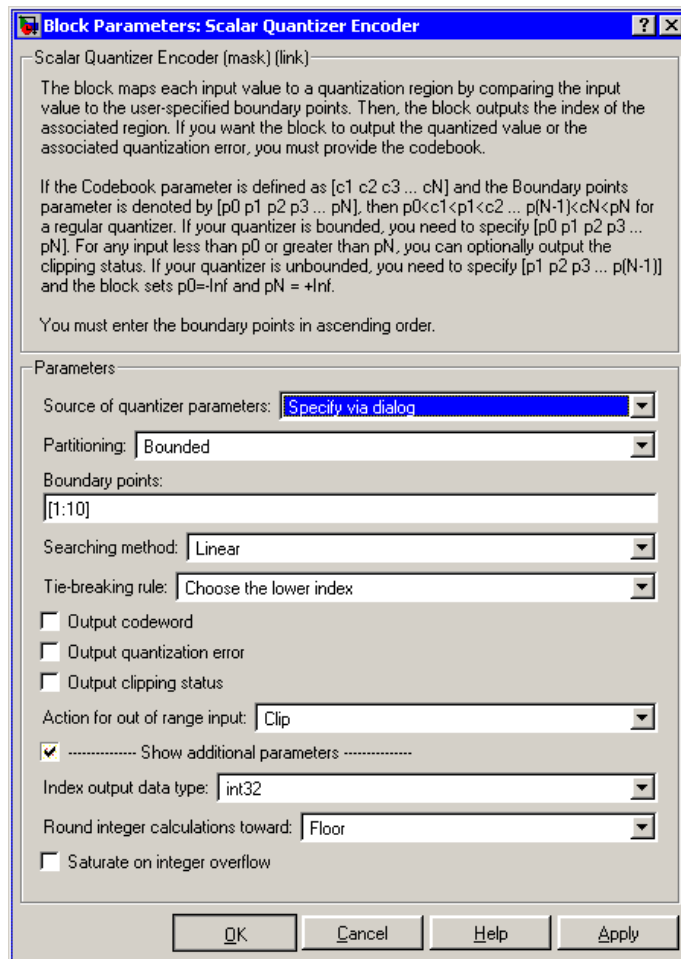
Index output data type

Specify the data type of the index output from the block at port I. You can choose int8, uint8, int16, uint16, int32, or uint32. Nontunable.

Round integer calculations toward

Select the rounding mode for integer output.

Scalar Quantizer Encoder

Saturate on integer overflow
When selected, overflows saturate.

References   Gersho, A. and R. Gray. *Vector Quantization and Signal Compression*. Boston: Kluwer Academic Publishers, 1992.

Supported Data Types

Port	Supported Data Types
U	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed only)
	• 8-, 16-, and 32-bit signed integers
B	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed only)
	• 8-, 16-, and 32-bit signed integers
C	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed only)
	• 8-, 16-, and 32-bit signed integers
I	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
Q(U)	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed only)
	• 8-, 16-, and 32-bit signed integers

Port	Supported Data Types
Err	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
S	• Boolean

For more information on what data types are supported for each quantizer mode, see "Data Type Support" on page 10-917. To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Quantizer	Simulink
Scalar Quantizer Decoder	Signal Processing Blockset
Scalar Quantizer Design	Signal Processing Blockset
Uniform Encoder	Signal Processing Blockset
Uniform Decoder	Signal Processing Blockset

Short-Time FFT

Purpose   Compute nonparametric estimate of the spectrum using short-time, fast Fourier transform (FFT) method

Library   Transforms

Description

The Short-Time FFT block computes a nonparametric estimate of the spectrum. The block buffers, applies a window, and zero pads the input signal. Then, the block takes the FFT of the signal, transforming it into the frequency domain.

Connect your sample-based or frame-based, single-channel analysis window to the w(n) port. For the **Analysis window length** parameter, enter the length of the analysis window, W. When your analysis window is a sample-based signal, the block buffers it into a frame-based signal with frame length W. When your analysis window is a frame-based signal and its frame length is not W, the block buffers the signal so that its frame length is W.

Connect your sample-based or frame-based, single-channel or multichannel input signal to the x(n) port. After the block buffers and windows this signal, it zero-pads the signal before computing the FFT. For the **FFT length** parameter, enter the length to which the block pads the input signal. For the **Overlap between consecutive windows (in samples)** parameter, enter the number of samples to overlap each frame of the input signal.

The complex-valued, sample-based, single-channel or multichannel short-time FFT is output at port X(n,k).

The Short-Time FFT block supports real and complex floating-point and fixed-point signals.

Fixed-Point Data Types

The following diagram shows the data types used within the Short-Time FFT subsystem block for fixed-point signals.
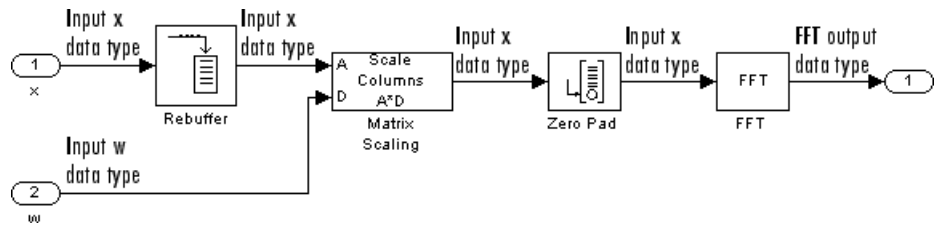
The settings for the fixed-point parameters of the Matrix Scaling block in the diagram above are as follows:

- Round integer calculations toward: `Floor`
- Saturate on integer overflow – unchecked
- Scaling vector – `Same word length as input`
- Product output – `Inherit via internal rule`
- Accumulator – `Inherit via internal rule`
- Output – `Same as first input`

The settings for the fixed-point parameters of the FFT block in the diagram above are as follows:

- Round integer calculations toward: `Floor`
- Saturate on integer overflow – unchecked
- Sine table – `Same word length as input`
- Product output – `Inherit via internal rule`
- Accumulator – `Inherit via internal rule`
- Output – `Inherit via internal rule`

Refer to the FFT and Matrix Scaling block reference pages for more information.

Short-Time FFT

Examples   The dspstsa_win32 demo illustrates how to use the Short-Time FFT and Inverse Short-Time FFT blocks to remove the background noise from a speech signal.

Dialog Box
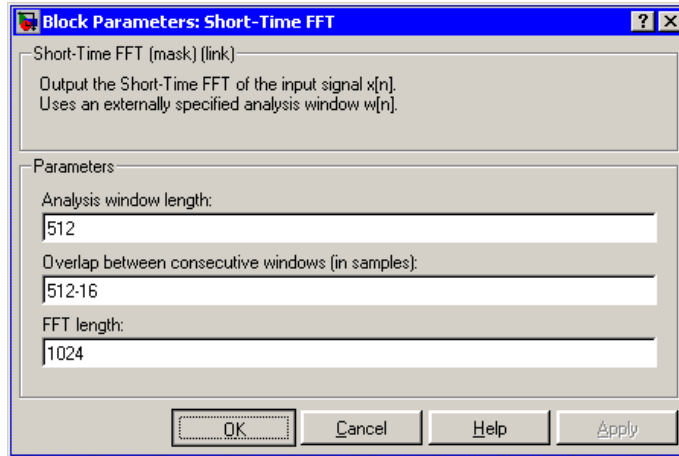
Analysis window length
    Enter the frame length of the analysis window.

Overlap between consecutive windows (in samples)
    Enter the number of samples of overlap for each frame of the input signal.

FFT length
    Enter the length to which the block pads the input signal.

References   Quatieri, Thomas E. *Discrete-Time Speech Signal Processing.* Englewood Cliffs, NJ: Prentice-Hall, 2001.

Supported Data Types

Port	Supported Data Types
x(n)	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
w(n)	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
X(n,k)	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Burg Method	Signal Processing Blockset
Inverse Short-Time FFT	Signal Processing Blockset
Magnitude FFT	Signal Processing Blockset
Periodogram	Signal Processing Blockset
Spectrum Scope	Signal Processing Blockset
Window Function	Signal Processing Blockset

Short-Time FFT

Yule-Walker Method	Signal Processing Blockset
`pwelch`	Signal Processing Toolbox

See "Power Spectrum Estimation" on page 6-6 for related information.

Purpose   Import signal from MATLAB workspace

Library   Signal Processing Sources

Description The Signal From Workspace block imports a signal from the MATLAB
workspace into the Simulink model. The **Signal** parameter specifies the
name of a MATLAB workspace variable containing the signal to import,
or any valid MATLAB expression defining a matrix or 3-D array.

When the **Signal** parameter specifies an M-by-N matrix ($M \neq 1$), each
of the N columns is treated as a distinct channel. You specify the
frame size in the **Samples per frame** parameter, M_o, and the output
is an M_o-by-N matrix containing M_o consecutive samples from each
signal channel. You specify the output sample period in the **Sample
time** parameter, T_s, and the output frame period is $M_o * T_s$. For $M_o = 1$,
the output is sample based; otherwise the output is frame based. For
convenience, an imported row vector ($M = 1$) is treated as a single
channel, so the output dimension is M_o-by-1.

When the **Signal** parameter specifies an M-by-N-by-P array, each of the
P pages (an M-by-N matrix) is output in sequence with period T_s. The
Samples per frame parameter must be set to 1, and the output is
always sample based.

Initial and Final Conditions

Unlike the Simulink From Workspace block, the Signal From Workspace
block holds the output value constant between successive output frames
(that is, no linear interpolation takes place). Additionally, the initial
signal values are always produced immediately at *t*=0.

When the block has output all of the available signal samples, it can
start again at the beginning of the signal, or simply repeat the final
value or generate zeros until the end of the simulation. (The block does
not extrapolate the imported signal beyond the last sample.) The **Form
output after final data value by** parameter controls this behavior:

Signal From Workspace

- When you specify `Setting To Zero`, the block generates zero-valued outputs for the duration of the simulation after generating the last frame of the signal.

- When you specify `Holding Final Value`, the block repeats the final sample for the duration of the simulation after generating the last frame of the signal.

- When you specify `Cyclic Repetition`, the block repeats the signal from the beginning after it reaches the last sample in the signal. If the frame size you specify in the **Samples per frame** parameter does not evenly divide the input length, a buffer block is inserted into the Signal From Workspace subsystem, and the model becomes multirate. If you do not want your model to become multirate, make sure the frame size evenly divides the input signal length.

  Select the **Warn when frame size does not evenly divide input length** parameter to be alerted when the input length is not an integer multiple of the frame size and your model will become multirate. Use the Model Explorer to turn these warnings on or off model-wide:

  a Select **Model Explorer** from the **View** menu in your model.

  b In the **Search** bar of the Model Explorer, search by `Property Name` for the `ignoreOrWarnInputAndFrameLengths` property. Each block with the **Warn when frame size does not evenly divide input length** check box appears in the list in the **Contents** pane.

  c Select each of the blocks for which you wish to toggle the warning parameter, and select or deselect the check box in the `ignoreOrWarnInputAndFrameLengths` column.

Examples   **Example 1**

In the first model below, the Signal From Workspace imports a two-channel signal from the workspace matrix A. The **Sample time** is set to 1 and the **Samples per frame** is set to 4, so the output is frame based with a frame size of 4 and a frame period of 4 seconds. The **Form**

output after final data value by parameter specifies `Setting To Zero`, so all outputs after the third frame (at *t*=8) are zero.
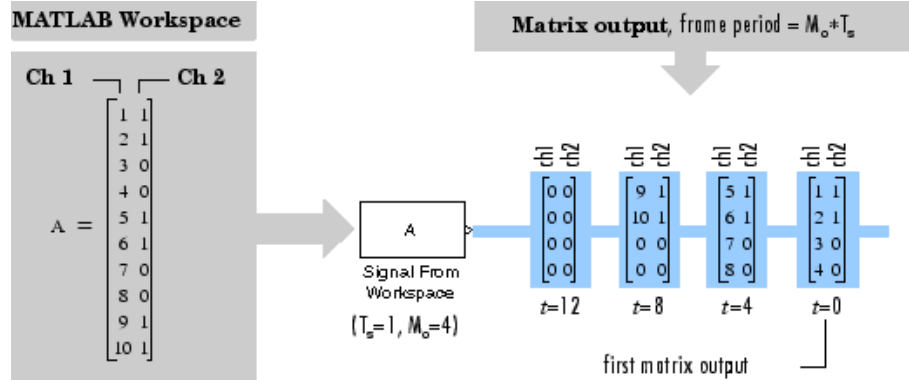
Example 2

In the second model below, the Signal From Workspace block imports a sample-based matrix signal from the 3-D workspace array A. Again, the **Form output after final data value by** parameter specifies `Setting To Zero`, so all outputs after the third (at *t*=2) are zero.
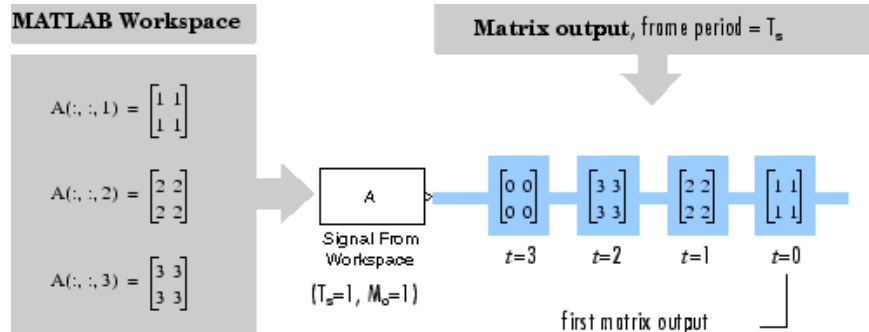
The **Samples per frame** parameter is set to 1 for 3-D input.

Signal From Workspace
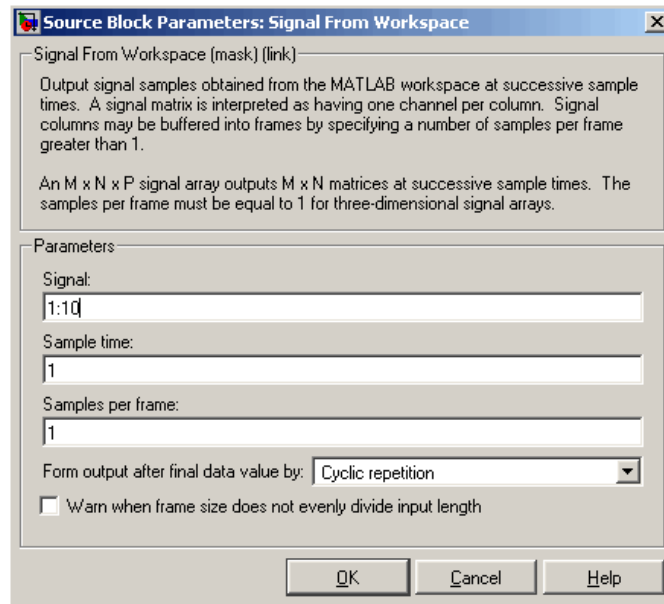
**Dialog
Box**

Signal

The name of the MATLAB workspace variable from which to import the signal, or a valid MATLAB expression specifying the signal.

Sample time

The sample period, T_s, of the output. The output frame period is M_o*T_s.

Samples per frame

The number of samples, M_o, to buffer into each output frame. This value must be 1 when you specify a 3-D array in the **Signal** parameter.

Form output after final data value by

Specifies the output after all of the specified signal samples have been generated. The block can output zeros for the duration of the simulation (Setting to zero), repeat the final data sample

(Holding Final Value) or repeat the entire signal from the beginning (Cyclic Repetition).

Warn when frame size does not evenly divide input length
Select this parameter to be alerted when the input length is not an integer multiple of the frame size and your model will become multirate. For more information, refer to "Initial and Final Conditions" on page 10-947.

This parameter is only visible when Cyclic Repetition is selected for the **Form output after final data value by** parameter.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point (signed and unsigned)
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

From Wave Device	Signal Processing Blockset
From Wave File	Signal Processing Blockset
Signal From Workspace	Signal Processing Blockset
From Workspace	Simulink
To Workspace	Simulink
Triggered Signal From Workspace	Signal Processing Blockset

Signal From Workspace

See the sections below for related information:

- "Creating Sample-Based Signals" on page 1-19
- "Creating Frame-Based Signals" on page 1-25
- "Importing and Exporting Sample-Based Signals" on page 1-55
- "Importing and Exporting Frame-Based Signals" on page 1-66

Purpose   Write simulation data to array in MATLAB workspace

Library   Signal Processing Sinks
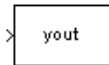
Description   The Signal To Workspace block writes data from your simulation into
an array in the MATLAB main workspace. The output array can be
2-D or 3-D, depending on whether the data is 1-D, sample based, or
frame based. The Signal To Workspace block and the Simulink To
Workspace block can output the same arrays when their parameters
are set appropriately.

For more information on the Signal To Workspace block, see the
following sections of this reference page:

- "Parameter Descriptions" on page 10-953

- "Output Dimension Summary" on page 10-955

- "Matching the Outputs of Signal To Workspace and To Workspace
  Blocks" on page 10-955

- "Examples" on page 10-956

Parameter Descriptions

The **Variable name** parameter is the name of the array in the
MATLAB workspace into which the block logs the simulation data.
The array is created in the workspace only after the simulation stops
running. When you enter the name of an existing workspace variable,
the block overwrites the variable with an array of simulation data after
the simulation stops running.

When the block input is sample based or 1-D, the **Limit data points
to last** parameter indicates how many *samples of data* to save. When
the block input is frame based, this parameter indicates how many
frames of data to save. When the simulation generates more than the
specified maximum number of samples or frames, the simulation saves
only the most recently generated data. To capture all data, set **Limit
data points to last** to `inf`.

Signal To Workspace

The **Decimation** parameter is the decimation factor. It can be set to any positive integer d, and allows you to write data at every dth sample. The default decimation, 1, writes data at every time step.

The **Frames** parameter sets the dimension of the output array to 2-D or 3-D for frame-based inputs. The block ignores this parameter for 1-D and sample-based inputs. The **Frames** parameter has the following two settings:

- `Log frames separately (3-D array)`: Given an M-by-N frame-based input signal, the block outputs an M-by-N-by-K array, where K is the number of frames logged by the end of the simulation. (K is bounded above by the **Limit data points to last** parameter.) Each input frame is an element of the 3-D array. (See "Example 2: Frame-Based Inputs" on page 10-957.)

- `Concatenate frames (2-D array)`: Given an M-by-N frame-based input signal with frame size f, the block outputs a (K*f)-by-N matrix, where K*f is the number of samples acquired by the end of the simulation. Each input frame is vertically concatenated to the previous frame to produce the 2-D array output. (See "Example 2: Frame-Based Inputs" on page 10-957.)

Signal to Workspace always logs sample-based input data as 3-D arrays, regardless of the **Frame** parameter setting. Given an M-by-N sample-based signal, the block outputs an M-by-N-by-L array, where L is the number of samples logged by the end of the simulation (L is bounded above by the **Limit data points to last** parameter). Each sample-based matrix is an element of the 3-D array. (See "Example 1: Sample-Based Inputs" on page 10-956.)

For 1-D vector inputs, the block outputs a 2-D matrix regardless of the setting of **Frame**. For a length-N 1-D vector input, the block outputs an L-by-N matrix. Each input vector is a row of the output matrix, vertically concatenated to the previous vector.

Output Dimension Summary

The following table summarizes the output array dimensions for various block inputs. In the table, *f* is the frame size of the input, K is the number of *frames* acquired by the end of the simulation, and L is the number of *samples* acquired by the end of the simulation (K and L are bounded above by the **Limit data points to last** parameter).

Input Signal Type	Signal To Workspace Output Dimension
Sample-based M-by-N matrix	M-by-N-by-L array
Length-N 1-D vector	L-by-N matrix
Frame-based M-by-N matrix; **Frame** set to `Log frames separately` (3-D array)	M-by-N-by-K array
Frame-based M-by-N matrix; **Frame** set to `Concatenate frames` (2-D array)	(K*f)-by-N matrix K*f is the number of samples acquired by the end of the simulation.

Matching the Outputs of Signal To Workspace and To Workspace Blocks

The To Workspace block in the Simulink Sinks Library and the Signal To Workspace block can output the same array when they are given the same inputs. To match the blocks' outputs, set their parameters as follows.

Signal To Workspace

Block Parameters	Signal To Workspace	To Workspace
Limit data points to last	x (any positive integer or inf)	x
Decimation	y (any positive integer, not inf)	y
Sample Time	No such parameter	-1
Save format	No such parameter	Array
Frames	Concatenate frames (2-D array)	No such parameter

Examples

Example 1: Sample-Based Inputs

In the following model, the input to the Signal To Workspace block is a 2-by-2 sample-based matrix signal with a sample time of 1 (generated by a Signal From Workspace block). The Signal To Workspace block logs 11 samples by the end of the simulation, and creates a 2-by-2-by-11 array, A, in the MATLAB workspace.
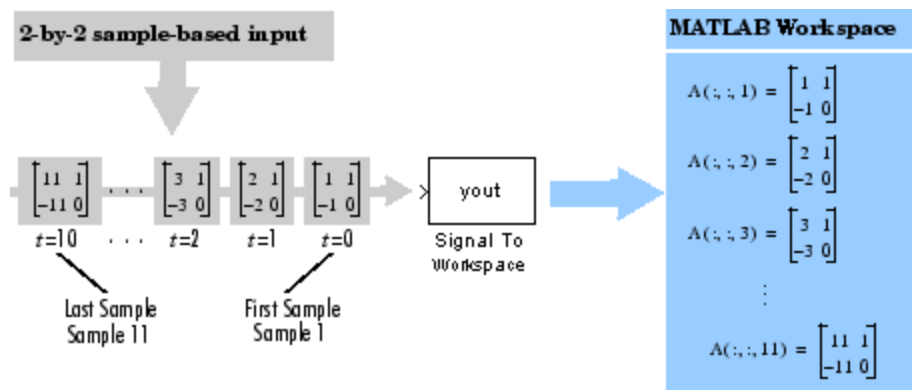
The block settings are as follows.

Signal To Workspace Block Parameters	
Variable name	yout
Limit data points to last	inf
Decimation	1
Frames	ignored since block input is not frame based
Configuration Dialog Box Parameters	
Start time	0
Stop time	10
Signal From Workspace Parameters (provides Signal To Workspace input)	
Signal	input1 (defined below)
Sample time	1
Samples per frame	1
Form output after final data value by	Setting to zero

```
input1 = cat(3, [1 1; -1 0], [2 1; -2 0],...,[11 1; -11 0])
```

Example 2: Frame-Based Inputs

In the following model, the input to the Signal To Workspace block is a 2-by-4 frame-based matrix signal with a frame period of 1 (generated by a Signal From Workspace block). The block logs 11 frames (two samples per frame) by the end of the simulation. The frames are concatenated to create a 22-by-4 matrix, A, in the MATLAB workspace.

The block settings for the following model are similar to the settings used in Example 1, except **Frames** is set to Concatenate frames (2-D array) and the Signal From Workspace parameter, **Signal**, is set to input2, where

Signal To Workspace

```
input2 = [1 -1 1 0; 2 -2 1 0; 3 -3 1 0;...; 22 -22 1 0]
```
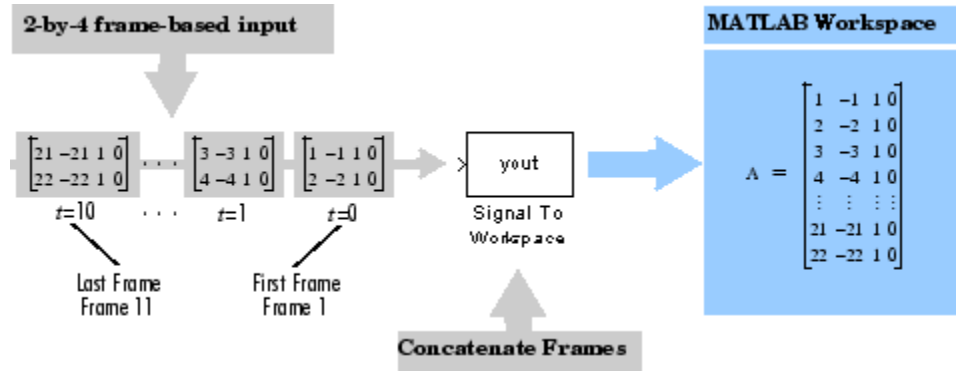

In the 2-D output, there is no indication of where one frame ends and another begins. By setting **Frames** to `Log frames separately (3-D array)` in this model, you can easily see each frame in the MATLAB workspace, as illustrated in the following model. Each of the 11 frames is logged separately to create a 2-by-4-by-11 array, A, in the MATLAB workspace.

Dialog Box

Block Parameters: Signal To Workspace [?] [X]

Signal To Workspace (mask) (link)

Write input to specified array in MATLAB's main workspace. Data is not available until the simulation is stopped or paused.

Parameters

Variable name:
yout

Limit data points to last:
inf

Decimation:
1

Frames: Concatenate frames (2-D array) ▼

☐ Log fixed-point data as a fi object

OK   Cancel   Help   Apply
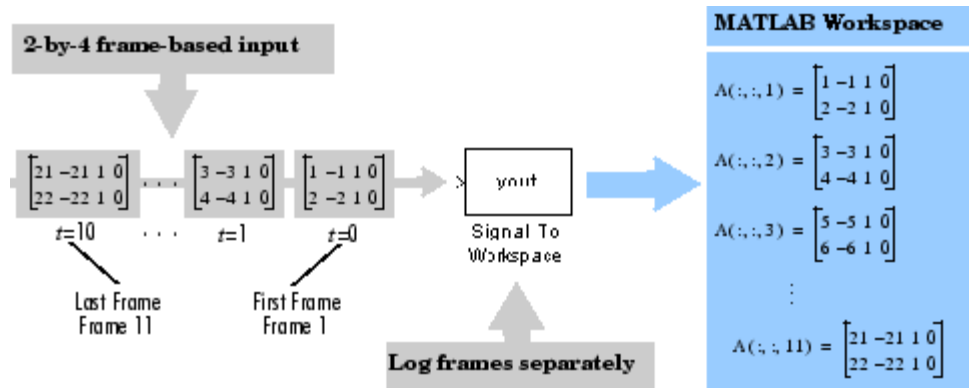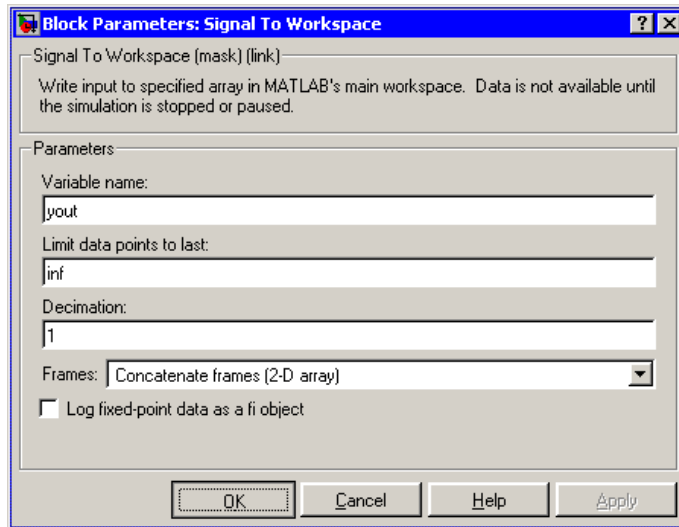
Variable name

The name of the array that holds the input data. Nontunable.

Limit data points to last

The maximum number of input samples (for sample-based inputs) or input frames (for frame-based inputs) to be saved. Nontunable.

Decimation

The decimation factor, d. Data is written at every dth sample. Nontunable.

Frames

The output dimensionality for frame-based inputs. **Frames** can be set to Concatenate frames (2-D array) or Log frames separately (3-D array). This parameter is ignored when inputs are not frame based. Nontunable.

Log fixed-point data as a fi object

Select to log fixed-point data to the MATLAB workspace as a fi object of the Fixed-Point Toolbox. Otherwise, fixed-point data is logged to the workspace as double.

Signal To Workspace

Supported Data Types

- Double-precision floating point

- Single-precision floating point

- Fixed point (signed and unsigned)

- 8-, 16-, and 32-bit signed integers

- 8-, 16-, and 32-bit unsigned integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Triggered To Workspace	Signal Processing Blockset
To Workspace	Simulink

Purpose	Generate continuous or discrete sine wave
Library	Signal Processing Sources

Description

The Sine Wave block generates a multichannel real or complex sinusoidal signal, with independent amplitude, frequency, and phase in each output channel. A real sinusoidal signal is generated when the **Output complexity** parameter is set to Real, and is defined by an expression of the type

$$y = A\sin(2\pi ft + \phi)$$

where you specify A in the **Amplitude** parameter, f in hertz in the **Frequency** parameter, and [[PHI]] in radians in the **Phase offset** parameter. A complex exponential signal is generated when the **Output complexity** parameter is set to Complex, and is defined by an expression of the type

$$y = Ae^{j(2\pi ft + \phi)} = A\{\cos(2\pi ft + \phi) + j\sin(2\pi ft + \phi)\}$$

Sections of This Reference Page

Generating Multichannel Outputs

For both real and complex sinusoids, the **Amplitude**, **Frequency**, and **Phase offset** parameter values (A, f, and [[PHI]]) can be scalars or length-N vectors, where N is the desired number of channels in the output. When you specify at least one of these parameters as a length-N vector, scalar values specified for the other parameters are applied to every channel.

For example, to generate the three-channel output containing the real sinusoids below, set **Output complexity** to Real and the other parameters as follows:

- **Amplitude** = [1 2 3]

- **Frequency** = [1000 500 250]

- **Phase offset** = [0 0 pi/2]

$$y = \begin{cases} \sin(2000\pi t) & \text{(channel 1)} \\[2ex] 2\sin(1000\pi t) & \text{(channel 2)} \\[2ex] 3\sin\left(500\pi t + \frac{\pi}{2}\right) & \text{(channel 3)} \end{cases}$$

Output Sample Time and Samples Per Frame

In all discrete modes, the block buffers the sampled sinusoids into frames of size M, where you specify M in the **Samples per frame** parameter. The output is a frame-based M-by-N matrix with frame period $M*T_s$, where you specify T_s in the **Sample time** parameter. For M=1, the output is sample based.

Sample Mode

The **Sample mode** parameter specifies the block's sampling property, which can be Continuous or Discrete:

- Continuous

  In continuous mode, the sinusoid in the ith channel, y_i, is computed as a continuous function,

  $$y_i = A_i \sin(2\pi f_i t + \phi_i) \qquad \text{(real)}$$

  or

  $$y_i = A_i e^{j(2\pi f_i t + \phi_i)} \qquad \text{(complex)}$$

  and the block's output is continuous. In this mode, the block's operation is the same as that of a Simulink Sine Wave block with **Sample time** set to 0. This mode offers high accuracy, but requires trigonometric function evaluations at each simulation step, which is computationally expensive. Additionally, because this method tracks absolute simulation time, a discontinuity will eventually occur when the time value reaches its maximum limit.

  Note also that many blocks in the Signal Processing Blockset do not accept continuous-time inputs.

- Discrete

  In discrete mode, the block's discrete-time output can be generated by directly evaluating the trigonometric function, by table lookup, or by a differential method. The three options are explained below.

Discrete Computational Methods

When you select Discrete from the **Sample mode** parameter, the secondary **Computation method** parameter provides three options for generating the discrete sinusoid:

- Trigonometric Fcn

- Table Lookup

- Differential

Trigonometric Fcn

The trigonometric function method computes the sinusoid in the ith channel, y_i, by sampling the continuous function

$$y_i = A_i \sin(2\pi f_i t + \phi_i) \qquad (\text{real})$$

or

$$y_i = A_i e^{j(2\pi f_i t + \phi_i)} \qquad (\text{complex})$$

with a period of T_s, where you specify T_s in the **Sample time** parameter. This mode of operation shares the same benefits and liabilities as the `Continuous` sample mode described above.

At each sample time, the block evaluates the sine function at the appropriate time value *within the first cycle* of the sinusoid. By constraining trigonometric evaluations to the first cycle of each sinusoid, the block avoids the imprecision of computing the sine of very large numbers, and eliminates the possibility of discontinuity during extended operations (when an absolute time variable might overflow). This method therefore avoids the memory demands of the table lookup method at the expense of many more floating-point operations.

Table Lookup

The table lookup method precomputes the *unique* samples of every output sinusoid at the start of the simulation, and recalls the samples from memory as needed. Because a table of finite length can only be constructed when all output sequences repeat, the method requires that the period of every sinusoid in the output be evenly divisible by the sample period. That is, $1/(f_i T_s) = k_i$ must be an integer value for every channel $i = 1, 2, ..., N$. When the **Optimize table for** parameter is set to `Speed`, the table constructed for each channel contains k_i elements. When the **Optimize table for** parameter is set to `Memory`, the table constructed for each channel contains $k_i/4$ elements.

For long output sequences, the table lookup method requires far fewer floating-point operations than any of the other methods, but can demand considerably more memory, especially for high sample

rates (long tables). This is the recommended method for models that are intended to emulate or generate code for DSP hardware, and that therefore need to be optimized for execution speed.

Differential

The differential method uses an incremental algorithm. This algorithm computes the output samples based on the output values computed at the previous sample time (and precomputed update terms) by making use of the following identities.

$$\sin(t + T_s) = \sin(t)\cos(T_s) + \cos(t)\sin(T_s)$$
$$\cos(t + T_s) = \cos(t)\cos(T_s) - \sin(t)\sin(T_s)$$

The update equations for the sinusoid in the ith channel, y_i, can therefore be written in matrix form as

$$\begin{bmatrix} \sin\{2\pi f_i(t + T_s) + \phi_i\} \\ \cos\{2\pi f_i(t + T_s) + \phi_i\} \end{bmatrix} = \begin{bmatrix} \cos(2\pi f_i T_s) & \sin(2\pi f_i T_s) \\ -\sin(2\pi f_i T_s) & \cos(2\pi f_i T_s) \end{bmatrix} \begin{bmatrix} \sin(2\pi f_i t + \phi_i) \\ \cos(2\pi f_i t + \phi_i) \end{bmatrix}$$

where you specify T_s in the **Sample time** parameter. Since T_s is constant, the right-hand matrix is a constant and can be computed once at the start of the simulation. The value of $A_i\sin[2\pi f_i(t+T_s)+[[PHI]]_i]$ is then computed from the values of $\sin(2\pi f_i t+[[PHI]]_i)$ and $\cos(2\pi f_i t+[[PHI]]_i)$ by a simple matrix multiplication at each time step.

This mode offers reduced computational load, but is subject to drift over time due to cumulative quantization error. Because the method is not contingent on an absolute time value, there is no danger of discontinuity during extended operations (when an absolute time variable might overflow).

Examples   The dspsinecomp demo provides a comparison of all the available sine generation methods.

Sine Wave

Dialog Box

The **Main** pane of the Sine Wave block dialog appears as follows:
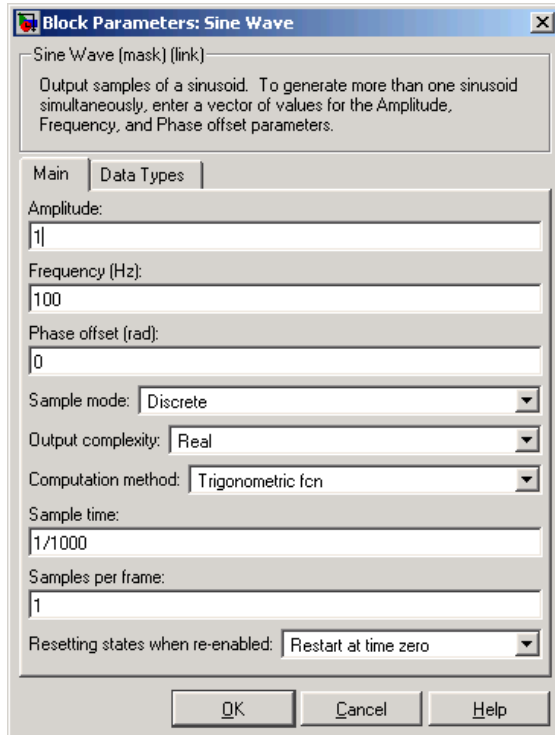
Opening this dialog box causes a running simulation to pause. See "Changing Source Block Parameters" in the online Simulink documentation for details.

Amplitude

A length-N vector containing the amplitudes of the sine waves in each of N output channels, or a scalar to be applied to all N channels. The vector length must be the same as that specified for the **Frequency** and **Phase offset** parameters. Tunable (when **Computation method** is *not* set to Table lookup); the

amplitude values can be altered while a simulation is running, but the vector length must remain the same.

Frequency

A length-N vector containing frequencies, in rad/s, of the sine waves in each of N output channels, or a scalar to be applied to all N channels. The vector length must be the same as that specified for the **Amplitude** and **Phase offset** parameters. You can specify positive, zero, or negative frequencies. Tunable (when **Computation method** is *not* set to `Table lookup`); the frequency values can be altered while a simulation is running, but the vector length must remain the same. This parameter is not tunable in the Simulink external mode when using the differential method.

Phase offset

A length-N vector containing the phase offsets, in radians, of the sine waves in each of N output channels, or a scalar to be applied to all N channels. The vector length must be the same as that specified for the **Amplitude** and **Frequency** parameters. This parameter is tunable when **Computation method** is *not* set to `Table lookup`; the phase values can be altered while a simulation is running, but the vector length must remain the same. This parameter is not tunable in the Simulink external mode when using the differential method.

Sample mode

The block's sampling behavior, `Continuous` or `Discrete`. This parameter is not tunable.

Output complexity

The type of waveform to generate: `Real` specifies a real sine wave, `Complex` specifies a complex exponential. This parameter is not tunable.

Computation method

The method by which discrete-time sinusoids are generated: `Trigonometric fcn`, `Table lookup`, or `Differential`. This parameter is not tunable. This parameter is disabled when

you select `Continuous` from the **Sample mode** parameter. For details, see "Discrete Computational Methods" on page 10-963.

Optimize table for

Optimizes the table of sine values for `Speed` or `Memory` (this parameter is only visible when the **Computation method** parameter is set to `Table lookup`). When optimized for speed, the table contains k elements, and when optimized for memory, the table contains $k/4$ elements, where k is the number of input samples in one full period of the sine wave.

Sample time

The period with which the sine wave is sampled, T_s. The block's output frame period is $M*T_s$, where you specify M in the **Samples per frame** parameter. This parameter is disabled when you select `Continuous` from the **Sample mode** parameter. This parameter is not tunable.

Samples per frame

The number of consecutive samples from each sinusoid to buffer into the output frame, M. This parameter is disabled when you select `Continuous` from the **Sample mode** parameter. Nontunable.

Resetting states when re-enabled

This parameter only applies when the Sine Wave block is located inside an enabled subsystem and the **States when enabling** parameter of the Enable block is set to `reset`. This parameter determines the behavior of the Sine Wave block when the subsystem is re-enabled. The block can either reset itself to its starting state (`Restart at time zero`), or resume generating the sinusoid based on the current simulation time (`Catch up to simulation time`). This parameter is disabled when you select `Continuous` from the **Sample mode** parameter.

The **Data types** pane of the Sine Wave block dialog appears as follows:
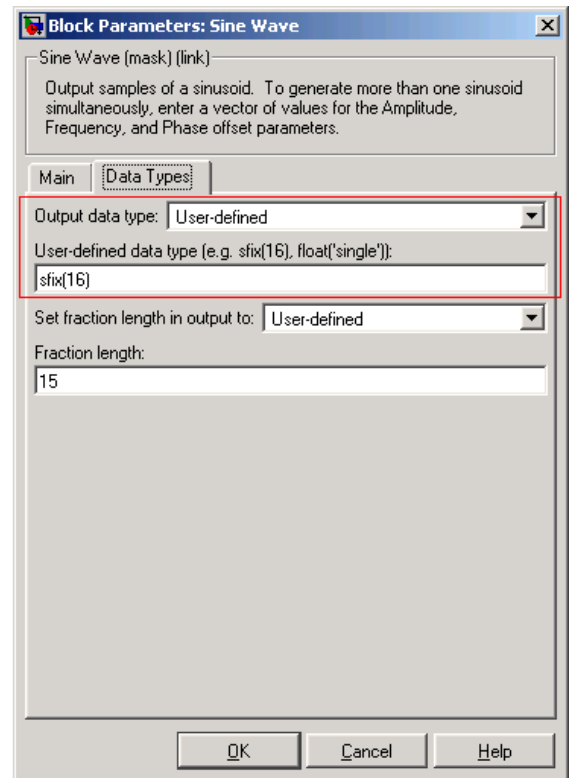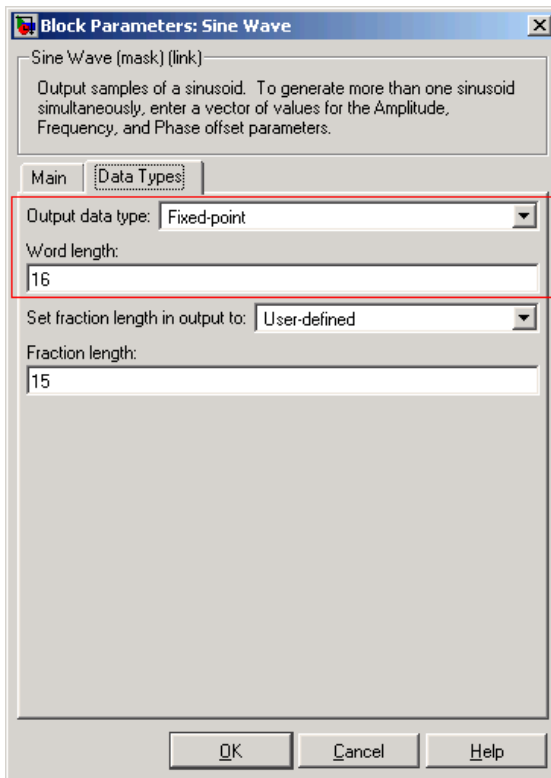
Output data type

Specify the output data type in out of the following ways:

Choose one of the built-in data types from the list.

Choose `Fixed-point` to specify the output data type and scaling in the **Word length**, **Set fraction length in output to**, and **Fraction length** parameters.

Choose `User-defined` to specify the output data type and scaling in the **User-defined data type**, **Set fraction length in output to**, and **Fraction length** parameters.

Choose `Inherit via back propagation` to set the output data type and scaling to match the next block downstream.

Word length
Specify the word length, in bits, of the fixed-point output data type. This parameter is only visible when you select `Fixed-point` for the **Output data type** parameter.

User-defined data type
Specify any built-in or fixed-point data type. You can specify fixed-point data types using the `sfix`, `ufix`, `sint`, `uint`, `sfrac`, and `ufrac` functions from Simulink Fixed Point. This parameter is only visible when you select `User-defined` for the **Output data type** parameter.

Set fraction length in output to
Specify the scaling of the fixed-point output by either of the following two methods:

Choose `Best precision` to have the output scaling automatically set such that the output signal has the best possible precision.

Choose `User-defined` to specify the output scaling in the **Fraction length** parameter.

This parameter is only visible when you select `Fixed-point` for the **Output data type** parameter, or when you select `User-defined` and the specified output data type is a fixed-point data type.

Fraction length
For fixed-point output data types, specify the number of fractional bits, or bits to the right of the binary point. This parameter is only visible when you select `Fixed-point` or `User-defined` for

the **Output data type** parameter and `User-defined` for the **Set fraction length in output to** parameter.

Supported Data Types	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed only)
	• 8-, 16-, and 32-bit signed integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Chirp	Signal Processing Blockset
Complex Exponential	Signal Processing Blockset
Signal From Workspace	Signal Processing Blockset
Signal Generator	Simulink
Sine Wave	Simulink
sin	MATLAB

Singular Value Decomposition

Purpose   Factor matrix using singular value decomposition

Library   Math Functions / Matrices and Linear Algebra / Matrix Factorizations
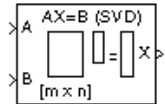
Description   The Singular Value Decomposition block factors the M-by-N input matrix A such that

$$A = U^+ diag(S) \cdot V^T$$

where U is an M-by-P matrix, V is an N-by-P matrix, S is a length-P vector, and P is defined as min(M,N).

When M = N, U and V are both M-by-M unitary matrices. When M > N, V is an N-by-N unitary matrix, and U is an M-by-N matrix whose columns are the first N columns of a unitary matrix. When N > M, U is an M-by-M unitary matrix, and V is an M-by-N matrix whose columns are the first N columns of a unitary matrix. In all cases, S is a 1-D vector of positive singular values having length P. The output is always sample based.

Length-N row inputs are treated as length-N columns.

```
[U,S,V] = svd(A,0)      % Equivalent MATLAB code for M > N
```

Note that the first (maximum) element of output S is equal to the 2-norm of the matrix A.

You can enable the U and V output ports by selecting the **Output the singular vectors** parameter.

**Dialog
Box**
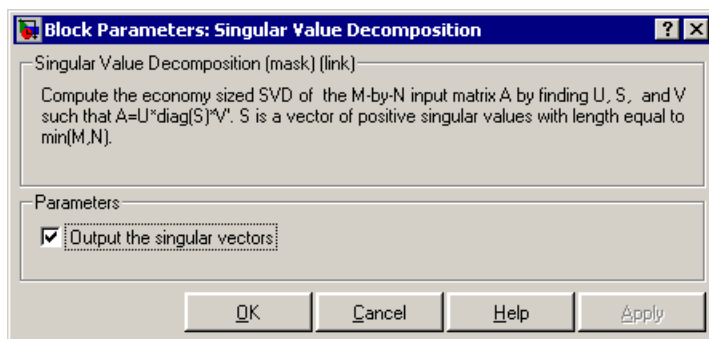
Output the singular vectors
> Enables the U and V output ports when selected.

References Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

**Supported
Data
Types**

- Double-precision floating point
- Single-precision floating point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Autocorrelation LPC	Signal Processing Blockset
Cholesky Factorization	Signal Processing Blockset
LDL Factorization	Signal Processing Blockset
LU Inverse	Signal Processing Blockset
Pseudoinverse	Signal Processing Blockset
QR Factorization	Signal Processing Blockset

Singular Value Decomposition

SVD Solver	Signal Processing Blockset
svd	MATLAB

See "Factoring Matrices" on page 6-9 for related information.

Purpose   Sort input elements by value

Library   Statistics

Description   The Sort block ranks the values of the input elements using either a quick sort or an insertion sort algorithm. The quick sort algorithm uses a recursive sort method and is faster at sorting more than 32 elements. The insertion sort algorithm uses a non-recursive method and is faster at sorting less than 32 elements. You should also always use the insertion sort algorithm when you are generating code from the Sort block if you do not want recursive function calls in your code. To specify the sort method, use the **Sort algorithm** parameter.

The **Mode** parameter specifies the block's mode of operation, and can be set to Value, Index, or Value and index.

The Sort block supports real and complex floating-point and fixed-point inputs. Signed and unsigned fixed-point signals are supported. The block output has the same signedness as the input.

Value Mode

When **Mode** is set to Value, the block sorts the elements in each column of the M-by-N input matrix u in order of ascending or descending value, as specified by the **Sort order** parameter.

```
val = sort(u)
val = flipud(sort(u))
```

For convenience, length-M 1-D vector inputs and *sample-based* length-M row vector inputs are both treated as M-by-1 column vectors.

The output at each sample time, val, is an M-by-N matrix containing the sorted columns of u. The output has the same frame status as the input.

Complex inputs are sorted by *magnitude squared*. For complex value u = a + bi, the magnitude squared is $a^2 + b^2$.

Sort

Index Mode

When **Mode** is set to Index, the block sorts the elements in each column of the M-by-N input matrix u,

```
[val,idx] = sort(u)
[val,idx] = flipud(sort(u))
```

and outputs the sample-based M-by-N index matrix, idx. The jth column of idx is an index vector that permutes the jth column of u to the desired sorting order.

```
val(:,j) = u(idx(:,j),j)
```

The index value outputs are always 32-bit unsigned integer values.

As in Value mode, length-M 1-D vector inputs and *sample-based* length-M row vector inputs are both treated as M-by-1 column vectors.

Value and Index Mode

When **Mode** is set to Value and index, the block outputs both the sorted matrix, val, and the index matrix, idx.

Fixed-Point Data Types

The parameters on the **Fixed-point** pane are only used for complex fixed-point inputs. Complex fixed-point inputs are sorted by magnitude squared. The sum of the squares of the real and imaginary parts of such an input are formed before a comparison is made, as described in "Value Mode" on page 10-975. The results of the squares of the real and imaginary parts are placed into the product output data type. The result of the sum of the squares is placed into the accumulator data type. These parameters are ignored for other types of inputs.

Dialog Box

The **Main** pane of the Sort block dialog appears as follows:
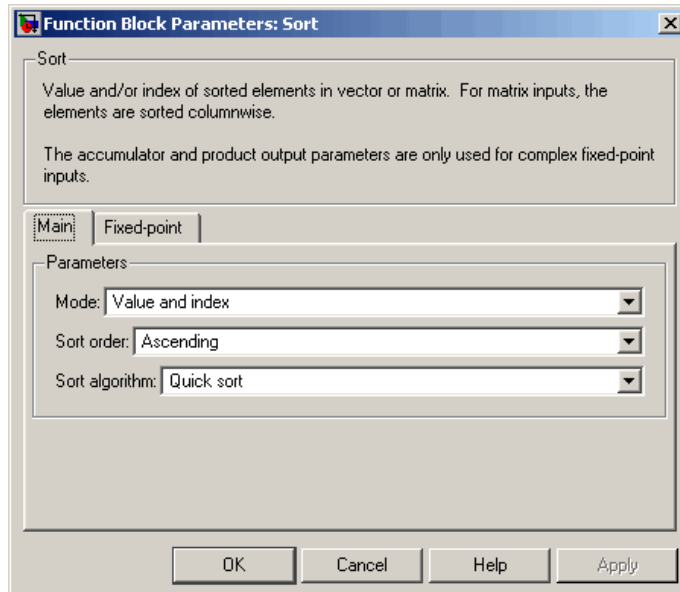
Mode

Specify the block's mode of operation: Output the sorted matrix (Value), the index matrix (Index), or both (Value and index).

Sort order

Specify the order in which to sort the training points, Descending or Ascending. Tunable, except in the Simulink external mode.

Sort algorithm

Specify whether the elements of the input are sorted using a Quick sort or an Insertion sort algorithm.

The **Fixed-point** pane of the Sort block dialog appears as follows:

Sort

Note The parameters on the **Fixed-point** pane are only used for complex fixed-point inputs. The sum of the squares of the real and imaginary parts of such an input are formed before a comparison is made, as described in "Value Mode" on page 10-975. The results of the squares of the real and imaginary parts are placed into the product output data type. The result of the sum of the squares is placed into the accumulator data type. These parameters are ignored for other types of inputs.

Rounding mode
   Select the rounding mode for fixed-point operations.

Overflow mode
   Select the overflow mode for fixed-point operations.

Product output

> Use this parameter to specify how you would like to designate
> the product output word and fraction lengths resulting from
> a complex-complex multiplication in the block. Refer to
> "Multiplication Data Types" on page 8-16 for more information:
>
> When you select Same as input, these characteristics will match
> those of the input to the block.
>
> When you select Binary point scaling, you are able to enter the
> word length and the fraction length of the product output, in bits.
>
> When you select Slope and bias scaling, you are able to enter
> the word length, in bits, and the slope of the product output. This
> block requires power-of-two slope and a bias of zero.

Accumulator

> Use this parameter to specify the accumulator word and fraction
> lengths resulting from a complex-complex multiplication in the
> block. Refer to "Multiplication Data Types" on page 8-16 for more
> information:
>
> When you select Same as product output, these characteristics
> will match those of the product output
>
> When you select Same as input, these characteristics will match
> those of the input to the block.
>
> When you select Binary point scaling, you are able to enter the
> word length and the fraction length of the accumulator, in bits.
>
> When you select Slope and bias scaling, you are able to enter
> the word length, in bits, and the slope of the accumulator. This
> block requires power-of-two slope and a bias of zero.

Sort

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• 8-, 16-, 32-, and 128-bit unsigned integers
	• 8-, 16-, 32-, and 128-bit signed integers
Val	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• 8-, 16-, 32-, and 128-bit unsigned integers
	• 8-, 16-, 32-, and 128-bit signed integers
Idx	• 32-bit unsigned integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Histogram	Signal Processing Blockset
Median	Signal Processing Blockset
sort	MATLAB

Purpose	Compute and display the periodogram of each input signal
Library	Signal Processing Sinks
Description	The Spectrum Scope block computes and displays the periodogram of the input. The input can be a 1-D vector or a 2-D matrix of any frame status.

Scope Properties Pane

When the block input is a 1-by-N sample-based vector or M-by-N sample-based matrix, you must select the **Buffer input** check box. Each of the N vector elements (or M*N matrix elements) is then treated as an independent channel, and the block buffers and displays the data in each channel independently.

When the input is frame based, you can leave the input as is, or rebuffer data by selecting the **Buffer input** check box and specifying the new buffer size. In the latter case, you can also specify an optional **Buffer overlap** parameter.

Buffering 1-D vector inputs is recommended. In this case, the block buffers the inputs into frames (the length of which are specified in the **Buffer size** parameter), where each 1-D input vector becomes a row in the buffered outcome. If 1-D vector input is left unbuffered, you will get a warning because the block is computing the FFT of a scalar. Though the scope window appears, it is unlikely you will see the plot. The block also displays a warning on the scope itself.

Use the **Buffer size** parameter, M_o to specify the number of input samples that the block buffers before computing and displaying the magnitude FFT. Use the **Buffer overlap** parameter, L, to specify the number of samples from the previous buffer to include in the current buffer. The number of new input samples the block acquires before computing and displaying the magnitude FFT is the difference between the buffer size and buffer overlap, M_o-L.

The display update period is $(M_o-L)*T_s$, where T_s is the input sample period. The display update period is equal to the input sample period when the **Buffer overlap** is M_o-1. For negative buffer overlap values,

Spectrum Scope

the block discards the appropriate number of input samples after the buffer fills, and updates the scope display at a slower rate than the zero-overlap case.

The **Window type** and **Window sampling** parameters apply to the specification of the window function; see the Window Function block reference page for more details on these parameters.

If you clear the **Specify FFT length** check box and the input is buffered, the block uses the buffer size as the FFT size. If you clear the check box and the input is not buffered, the block uses the input size as the FFT size. When you select the check box, the **FFT length** parameter, N_{fft}, appears on the dialog box. Enter the number of samples on which you want the block to perform the FFT. The block zero pads or truncates every channel's buffer to N_{fft} before computing the FFT.

The number of spectra to average is set by the **Number of spectral averages** parameter. Setting this parameter to 1 effectively disables averaging; see the Periodogram block reference page for more information.

Display Properties Pane

For information about these parameters, see "Display Properties Pane" on page 10-1192 of the Vector Scope block reference page.

Axis Properties Pane

The **Frequency units** parameter specifies whether the frequency axis values should be in units of Hertz or rad/s. When the **Frequency units** parameter specifies `Hertz`, the spacing between frequency points is $1/(N_{fft}T_s)$. For **Frequency units** of `rad/sec`, the spacing between frequency points is $2\pi/(N_{fft}T_s)$.

The **Frequency range** parameter specifies the range of frequencies over which the magnitudes in the input should be plotted. The available options are `[0..Fs/2]`, `[-Fs/2..Fs/2]`, and `[0..Fs]`, where F_s is the original time-domain signal's sample frequency.

Note that all of the FFT-based blocks in the Signal Processing Blockset, including those in the Power Spectrum Estimation library, compute

the FFT at frequencies in the range [0,F_s). The **Frequency range** parameter controls only the displayed range of the signal.

If you select the **Inherit sample increment from input** check box, the block computes the frequency data from the sample period of the input to the block. This is valid when the following conditions hold:

- The input to the block is the original signal, with no samples added or deleted (by insertion of zeros, for example).

- The sample period of the time-domain signal in the simulation is equal to the period with which the physical signal was originally sampled.

In cases where not all of these conditions hold, you should specify the appropriate value for the **Sample time of original time-series** parameter.

To correctly scale the horizontal (frequency) axis for frequency-domain signals, the block needs to know the actual sample period of the time-domain input. You specify this in the **Sample time of original time series** parameter, T_s.

The **Amplitude scaling** parameter allows you to select Magnitude or dB scaling along the *y*-axis.

Minimum Y-limit and **Maximum Y-limit** parameters set the range of the vertical axis.

The **Y-axis title** is the text to be displayed to the left of the y-axis.

Line Properties Pane

For information about these parameters, see "Line Properties Pane" on page 10-1196 of the Vector Scope block reference page.

Spectrum Scope

Dialog Box

Scope Properties Pane
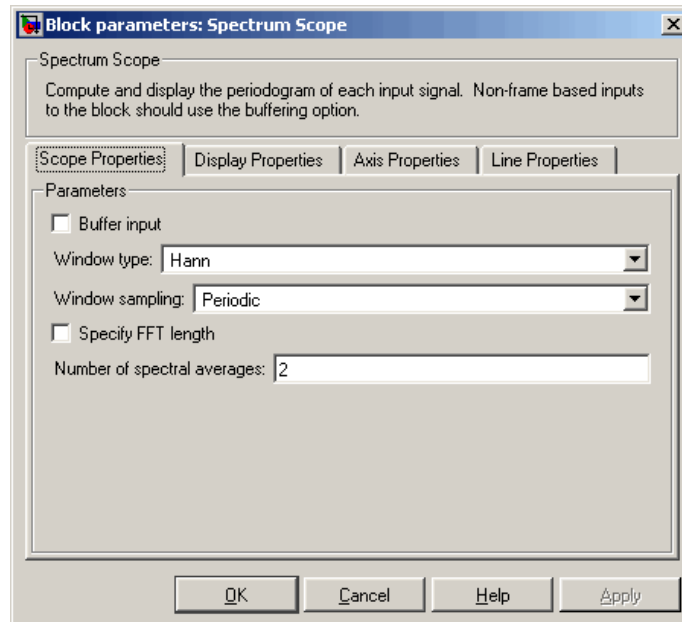
Buffer input
  Select this check box to rebuffer the input data.

Buffer size
  The number of signal samples to include in each buffer. This parameter is visible if you select the **Buffer input** check box.

Buffer overlap
  The number of samples by which consecutive buffers overlap. This parameter is visible if you select the **Buffer input** check box.

Window type
  Enter the type of window to apply. See the Window Function block reference page for more details. Tunable.

Stopband attenuation in dB

Enter the level, in dB, of stopband attenuation, R_s, for the Chebyshev window. This parameter is visible if, for the **Window type** parameter, you choose Chebyshev. Tunable.

Beta

Enter the β parameter for the Kaiser window. This parameter is visible if, for the **Window type** parameter, you chose Kaiser. Increasing **Beta** widens the mainlobe and decreases the amplitude of the window sidelobes in the window's frequency magnitude response. Tunable.

Window sampling

From the list, choose Symmetric or Periodic. This parameter is visible if, for the **Window type** parameter, you choose Blackman, Hamming, Hann, or Hanning. Tunable.

Specify FFT length

Select this check box to enter the FFT length.

FFT length

The number of samples on which to perform the FFT. When the **FFT length** differs from the buffer size, the data is zero-padded or truncated as needed. This parameter is visible if you select the **Specify FFT length** check box.

Number of spectral averages

The number of spectra to average. Setting this parameter to 1 effectively disables averaging. See the Periodogram block reference page for more information.

Display Properties Pane

Show grid

    Toggle the scope grid on and off. Tunable.

Persistence

    Select this check box to maintain successive displays. That is, the scope does not erase the display after each frame (or collection of frames), but overlays successive input frames in the scope display. Tunable.

Frame number

    If you select this check box, the number of the current frame in the input sequence appears in the Vector Scope window. Tunable.

Channel legend

    Toggles the legend on and off. Tunable.

Compact display

Resizes the scope to fill the window. Tunable.

Open scope at start of simulation

Select this check box to open the scope at the start of the simulation. When this parameter is cleared, the scope will not open automatically during the simulation. Tunable.

Open scope immediately

If the scope is not open during simulation, select this check box to open it. This parameter is visible only while the simulation is running.

Scope position

A four-element vector of the form `[left bottom width height]` specifying the position of the scope window. `(0,0)` is the lower-left corner of the display. Tunable.

Spectrum Scope

Axis Properties Pane
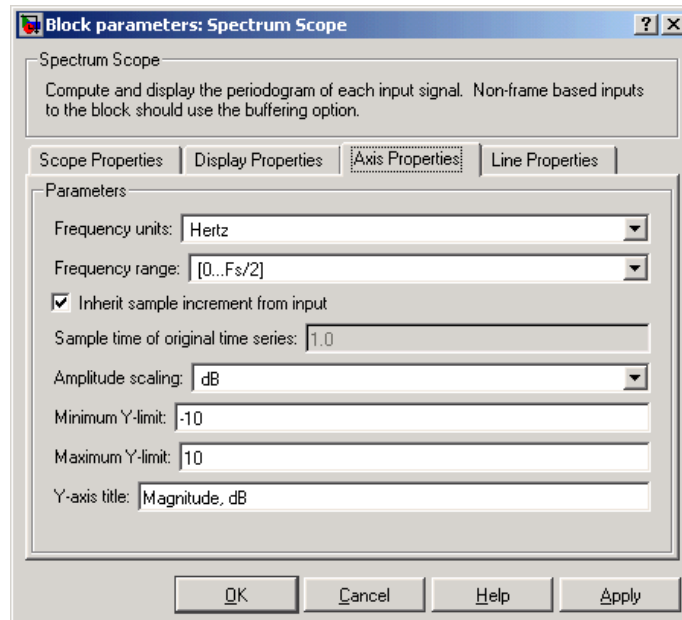
Frequency units

Choose the frequency units for the horizontal axis, Hertz or rad/sec. Tunable.

Frequency range

Specify the frequency range over which to plot the data. Tunable.

Inherit sample increment from input

If you select this check box, the block computes the time-domain sample period from the frame period and frame size of the frequency-domain input. Use this parameter only when the length of each frame of frequency-domain data is the same as the length of the frame of time-domain data from which it was generated. Tunable.

Sample time of original time series

Enter the sample period of the original time-domain signal. Tunable.

Amplitude scaling

Choose the scaling for the *y*-axis, dB or Magnitude. Tunable.

Minimum Y-limit

The minimum value of the *y*-axis. Tunable.

Maximum Y-limit

The maximum value of the *y*-axis. Tunable.

Y-axis title

The text to be displayed to the left of the *y*-axis. Tunable.

Line Properties Pane
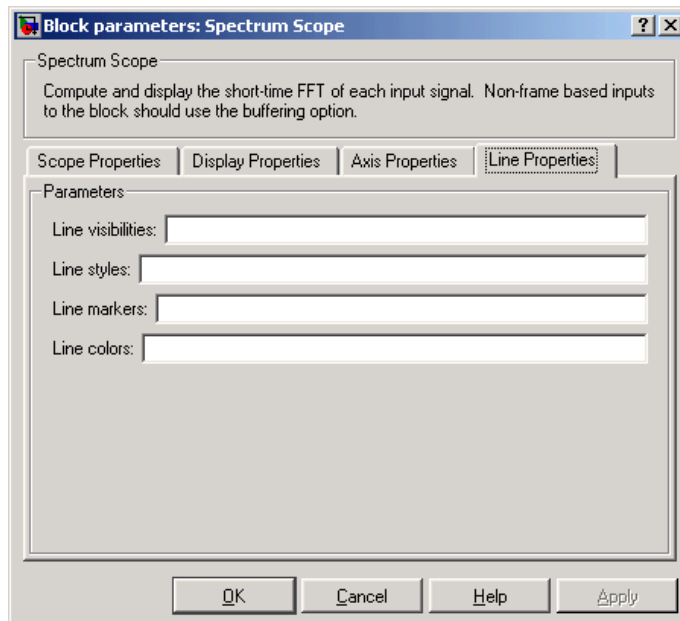
Line visibilities

Enter on or off to specify the visibility of the various channels' scope traces. Separate your choices for each channel with by a pipe (|) symbol. Tunable.

Line styles

Enter the line styles of the various channels' scope traces using the MATLAB line function LineStyle formats. Separate your choices for each channel with by a pipe (|) symbol. Tunable.

Line markers

Enter the line markers of the various channels' scope traces using the MATLAB line function Marker formats. Separate your choices for each channel with by a pipe (|) symbol. Tunable.

Line colors

Enter the colors of the various channels' scope traces using the MATLAB ColorSpec formats. Separate your choices for each channel with by a pipe (|) symbol. Tunable.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Custom data types
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

FFT	Signal Processing Blockset
Periodogram	Signal Processing Blockset
Short-Time FFT	Signal Processing Blockset
Vector Scope	Signal Processing Blockset

Stack

Purpose   Store inputs into LIFO register.

Library   Signal Management / Buffers

Description   The Stack block stores a sequence of input samples in a last in, first
out (LIFO) register. The register capacity is set by the **Stack depth**
parameter, and inputs can be scalars, vectors, or matrices.

The block *pushes* the input at the In port onto the top of the stack when
a trigger event is received at the Push port. When a trigger event is
received at the Pop port, the block *pops* the top element off the stack
and holds the Out port at that value. The last input to be pushed onto
the stack is always the first to be popped off.
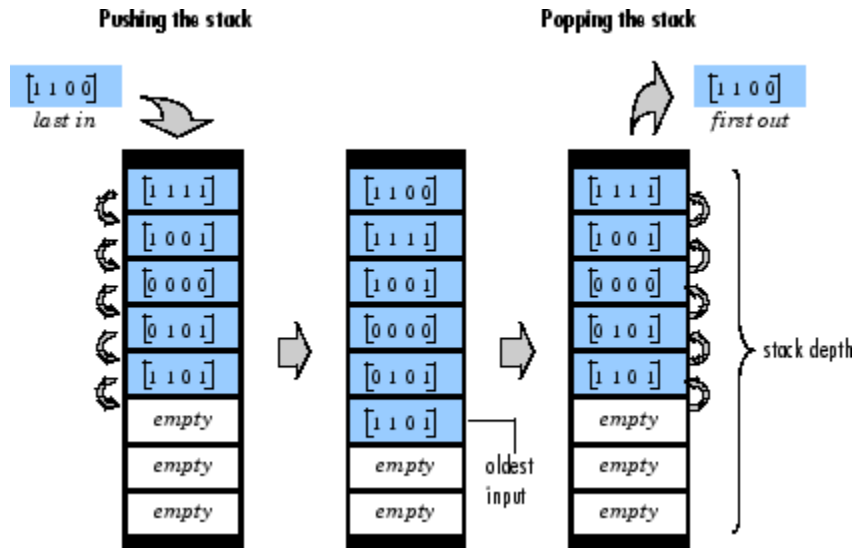
A trigger event at the optional Clr port (enabled by the **Clear input**
check box) empties the stack contents. When you select **Clear output
port on reset**, then a trigger event at the Clr port empties the stack
and sets the value at the Out port to zero. This setting also applies
when a disabled subsystem containing the Stack block is reenabled; the

Out port value is only reset to zero in this case when you select **Clear output port on reset**.

When two or more of the control input ports are triggered at the same time step, the operations are executed in the following order:

1 Clr

2 Push

3 Pop

The rate of the trigger signal must be the same as the rate of the data signal input. You specify the triggering event for the Push, Pop, and Clr ports in the **Trigger type** pop-up menu:

- Rising edge – Triggers execution of the block when the trigger input does one of the following:

    - Rises from a negative value to a positive value or zero

    - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)

- Falling edge – Triggers execution of the block when the trigger input does one of the following:

    - Falls from a positive value to a negative value or zero

- Falls from zero to a negative value, where the fall is not a
  continuation of a fall from a positive value to zero (see the following
  figure)

- `Either edge` – Triggers execution of the block when the trigger input
  is a `Rising edge` or `Falling edge` (as described above).

- `Non-zero sample` – Triggers execution of the block at each sample
  time that the trigger input is not zero.

Note When running simulations in the Simulink `MultiTasking`
mode, sample-based trigger signals have a one-sample latency, and
frame-based trigger signals have one frame of latency. Thus, there
is a one-sample or one-frame delay between the time the block
detects a trigger event, and when it applies the trigger. For more
information on latency and the Simulink tasking modes, see "Excess
Algorithmic Delay (Tasking Latency)" on page 2-56 and "Models with
Multiple Sample Rates" in the Real-Time Workshop User's Guide
documentation.

The **Push full stack** parameter specifies the block's behavior when a
trigger is received at the `Push` port but the register is full. The **Pop
empty stack** parameter specifies the block's behavior when a trigger is
received at the `Pop` port but the register is empty. The following options
are available for both cases:

- Ignore – Ignore the trigger event, and continue the simulation.

- Warning – Ignore the trigger event, but display a warning message in the MATLAB command window.

- Error – Display an error dialog box and terminate the simulation.

Note The **Push full stack** and **Pop empty stack** parameters are diagnostic parameters. Like all diagnostic parameters on the Configuration Parameters dialog box, they are set to Ignore in the Real-Time Workshop code generated for this block.

The **Push full stack** parameter additionally offers the **Dynamic reallocation** option, which dynamically resizes the register to accept as many additional inputs as memory permits. To find out how many elements are on the stack at a given time, enable the Num output port by selecting the **Output number of stack entries** option.

Examples

Example 1

The table below illustrates the Stack block's operation for a **Stack depth** of 4, **Trigger type** of Either edge, and **Clear output port on reset** enabled. Because the block triggers on both rising and falling edges in this example, each transition from 1 to 0 or 0 to 1 in the Push, Pop, and Clr columns below represents a distinct trigger event. A 1 in the Empty column indicates an empty buffer, while a 1 in the Full column indicates a full buffer.

In	Push	Pop	Clr	Stack	Out	Empty	Full	Num
1	0	0	0	*top* ▮□□□□ *bottom*	0	1	0	0
2	1	0	0	*top* ▮ 2 □□□ *bottom*	0	0	0	1
3	0	0	0	*top* ▮ 3 2 □□ *bottom*	0	0	0	2

In	Push	Pop	Clr	Stack	Out	Empty	Full	Num
4	1	0	0	top [4 | 3 | 2 |] bottom	0	0	0	3
5	0	0	0	top [5 | 4 | 3 | 2] bottom	0	0	1	4
6	0	1	0	top [4 | 3 | 2 |] bottom	5	0	0	3
7	0	0	0	top [3 | 2 | |] bottom	4	0	0	2
8	0	1	0	top [2 | | |] bottom	3	0	0	1
9	0	0	0	top [| | |] bottom	2	1	0	0
10	1	0	0	top [10 | | |] bottom	2	0	0	1
11	0	0	0	top [11 | 10 | |] bottom	2	0	0	2
12	1	0	1	top [12 | | |] bottom	0	0	0	1

Note that at the last step shown, the Push and Clr ports are triggered
simultaneously. The Clr trigger takes precedence, and the stack is first
cleared and then pushed.

Example 2

The dspqdemo demo provides an example of the related Queue block.

Dialog Box

Block Parameters: Stack

Stack (mask) (link)

Implements a stack, or "Last In, First Out" (LIFO) register.

Parameters

Stack depth:

`8`

Trigger type: `Rising edge`

Push full stack: `Dynamic reallocation`

Pop empty stack: `Warning`

☐ Empty stack output
☐ Full stack output
☐ Output number of stack entries
☐ Clear input
☐ Clear output port on reset

[<u>O</u>K] [<u>C</u>ancel] [<u>H</u>elp] [Apply]
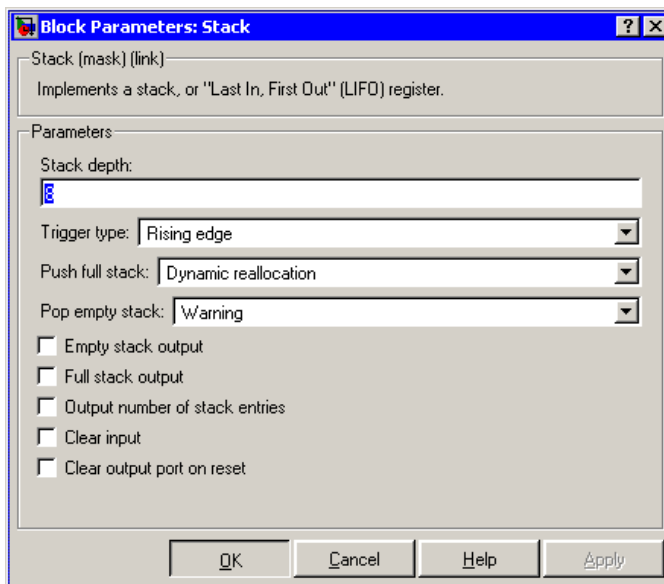
Stack depth

> The number of entries that the LIFO register can hold.

Trigger type

> The type of event that triggers the block's execution. The rate of the trigger signal must be the same as the rate of the data signal input. Tunable.

Push full stack

> Response to a trigger received at the Push port when the register is full. Inputs to this port must have the same built-in data type as inputs to the Pop and Clr input ports.

Pop empty stack

> Response to a trigger received at the Pop port when the register is empty. Inputs to this port must have the same built-in data type as inputs to the Push and Clr input ports. Tunable.

Empty stack output

Enable the `Empty` output port, which is high (`1`) when the stack is empty, and low (`0`) otherwise.

Full stack output

Enable the `Full` output port, which is high (`1`) when the stack is full, and low (`0`) otherwise. The `Full` port remains low when you select **Dynamic reallocation** from the **Push full stack** parameter.

Output number of stack entries

Enable the `Num` output port, which tracks the number of entries currently on the stack. When inputs to the `In` port are double-precision values, the outputs from the `Num` port are double-precision values. Otherwise, the outputs from the `Num` port are 32-bit unsigned integer values.

Clear input

Enable the `Clr` input port, which empties the stack when the trigger specified by the **Trigger type** is received. Inputs to this port must have the same built-in data type as inputs to the `Push` and `Pop` input ports.

Clear output port on reset

Reset the `Out` port to zero (in addition to clearing the stack) when a trigger is received at the `Clr` input port. Tunable.

**Supported
Data
Types**

Port	Supported Data Types
In	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Custom data types • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Push	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Custom data types • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers Inputs to this port must have the same built-in data type as inputs to the Pop and Clr input ports

Stack

Port	Supported Data Types
Pop	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Custom data types • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers Inputs to this port must have the same built-in data type as inputs to the Push and Clr input ports.
Clr	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Custom data types • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers Inputs to this port must have the same built-in data type as inputs to the Push and Pop input ports.

Port	Supported Data Types
Out	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Custom data types • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Empty	• Double-precision floating point • Boolean The block outputs Boolean values at this port when Boolean support is enabled, as described in "Effects of Enabling and Disabling Boolean Support" on page 7-17. To learn how to disable Boolean output support, see "Steps to Disabling Boolean Support" on page 7-18
Full	• Double-precision floating point • Boolean The block outputs Boolean values at this port when Boolean support is enabled, as described in "Effects of Enabling and Disabling Boolean Support" on page 7-17. To learn how to disable Boolean output support, see "Steps to Disabling Boolean Support" on page 7-18
Num	• Double-precision floating point The block outputs a double-precision floating-point value at this port when the data type of the In port is double-precision floating-point. • 32-bit unsigned integers

Stack

Port	Supported Data Types
	The block outputs a 32-bit unsigned integer value at this port when the data type of the In port is anything other than double-precision floating-point.

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Buffer	Signal Processing Blockset
Delay Line	Signal Processing Blockset
Queue	Signal Processing Blockset

Purpose   Find standard deviation of an input or sequence of inputs

Library   Statistics
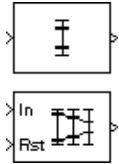
Description   The Standard Deviation block computes the standard deviation of each column in the input, or tracks the standard deviation of a sequence of inputs over a period of time. The **Running standard deviation** parameter selects between basic operation and running operation.

Basic Operation

When you do *not* select the **Running standard deviation** check box, the block computes the standard deviation of each column in M-by-N input matrix u independently at each sample time.

```
y = std(u)      % Equivalent MATLAB code
```

For convenience, length-M 1-D vector inputs and *sample-based* length-M row vector inputs are both treated as M-by-1 column vectors. (A scalar input generates a zero-valued output.)

The output at each sample time, y, is a 1-by-N vector containing the standard deviation for each column in u. For purely real or purely imaginary inputs, the standard deviation of the *j*th column is the square root of the variance

$$y_j = \sigma_j = \sqrt{\frac{\sum_{i=1}^{M} |u_{ij} - \mu_j|^2}{M-1}} \qquad 1 \le j \le N$$

where μ_j is the mean of *j*th column. For complex inputs, the output is the *total standard deviation* for each column in u, which is the square root of the *total variance* for that column.

$$\sigma_j = \sqrt{\sigma_{j, Re}^2 + \sigma_{j, In}^2}$$

Standard Deviation

Note that the total standard deviation is *not* equal to the sum of the real and imaginary standard deviations. The frame status of the output is the same as that of the input.

Running Operation

When you select the **Running standard deviation** check box, the block tracks the standard deviation of each channel in a *time-sequence* of M-by-N inputs. For sample-based inputs, the output is a sample-based M-by-N matrix with each element y_{ij} containing the standard deviation of element u_{ij} over all inputs since the last reset. For frame-based inputs, the output is a frame-based M-by-N matrix with each element y_{ij} containing the standard deviation of the *j*th column over all inputs since the last reset, up to and including element u_{ij} of the current input.

As in basic operation, length-M 1-D vector inputs and *sample-based* length-M row vector inputs are both treated as M-by-1 column vectors.

Resetting the Running Standard Deviation

The block resets the running standard deviation whenever a reset event is detected at the optional Rst port. The reset signal rate must be a positive integer multiple of the rate of the data signal input.

You specify the reset event in the **Reset port** parameter:

- None disables the Rst port.

- Rising edge – Triggers a reset operation when the Rst input does one of the following:

  - Rises from a negative value to a positive value or zero

  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)

- Falling edge – Triggers a reset operation when the Rst input does one of the following:

  - Falls from a positive value to a negative value or zero

  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero (see the following figure)

- Either edge – Triggers a reset operation when the Rst input is a Rising edge or Falling edge (as described above).

- Non-zero sample – Triggers a reset operation at each sample time that the Rst input is not zero.

Standard Deviation

Examples   The Standard Deviation block in the model below calculates the running
standard deviation of a frame-based 3-by-2 (two-channel) matrix
input, u. The running standard deviation is reset at $t=2$ by an impulse
to the block's Rst port.
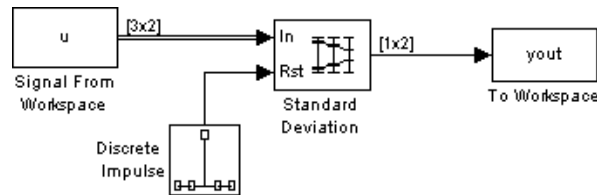
The Standard Deviation block has the following settings:

- **Running standard deviation** = ☑

- **Reset port** = Non-zero sample

The Signal From Workspace block has the following settings:

- **Signal** = u

- **Sample time** = 1/3

- **Samples per frame** = 3

where

```
u = [6 1 3 -7 2 5 8 0 -1 -3 2 1;1 3 9 2 4 1 6 2 5 0 4 17]'
```

The Discrete Impulse block has the following settings:

- **Delay (samples)** = 2
- **Sample time** = 1
- **Samples per frame** = 1

The block's operation is shown in the figure below.
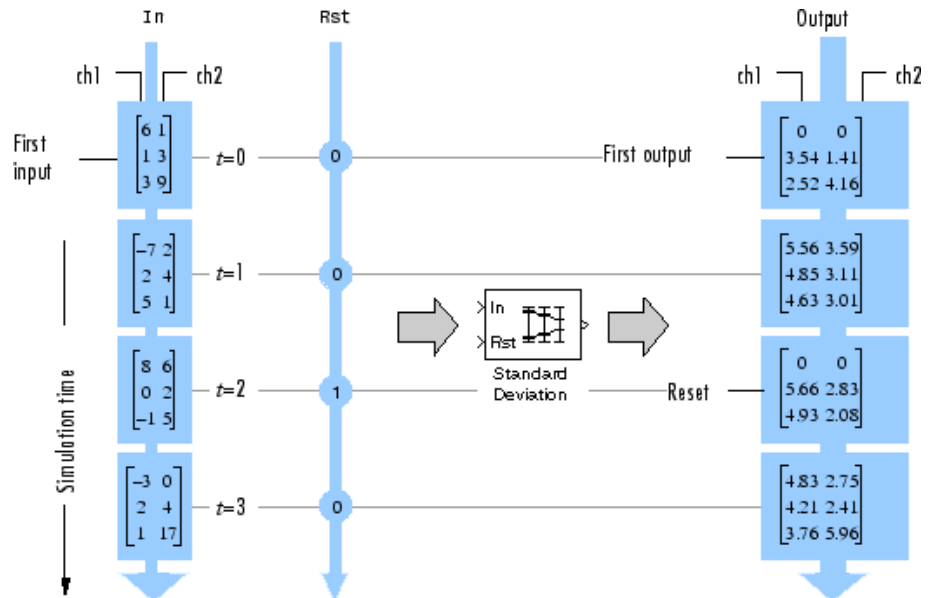
Standard Deviation

Dialog Box
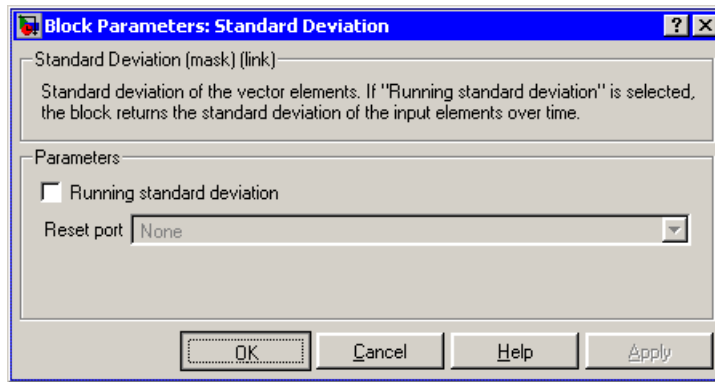
Running standard deviation
Enables running operation when selected.

Reset port
Determines the reset event that causes the block to reset the running standard deviation. The reset signal rate must be a positive integer multiple of the rate of the data signal input. This parameter is enabled only when you select **Running standard deviation**. For more information, see "Resetting the Running Standard Deviation" on page 10-1004.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Boolean – The block accepts Boolean inputs to the Rst port.

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Mean	Signal Processing Blockset
RMS	Signal Processing Blockset
Variance	Signal Processing Blockset
std	MATLAB

Submatrix

Purpose   Select subset of elements (submatrix) from matrix input

Library
- Math Functions / Matrices and Linear Algebra / Matrix Operations
- Signal Management / Indexing

Description
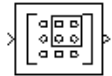
The Submatrix block extracts a contiguous submatrix from the M-by-N input matrix u. A length-M 1-D vector input is treated as an M-by-1 matrix. The **Row span** parameter provides three options for specifying the range of rows in u to be retained in submatrix output y:

- All rows

  Specifies that y contains all M rows of u.

- One row

  Specifies that y contains only one row from u. The **Starting row** parameter (described below) is enabled to allow selection of the desired row.

- Range of rows

  Specifies that y contains one or more rows from u. The **Row** and **Ending row** parameters (described below) are enabled to allow selection of the desired range of rows.

The **Column span** parameter contains a corresponding set of three options for specifying the range of columns in u to be retained in submatrix y: All columns, One column, or Range of columns. The One column option enables the **Column** parameter, and Range of columns options enable the **Starting column** and **Ending column** parameters.

The output has the same frame status as the input.

Range Specification Options

When you select One row or Range of rows from the **Row span** parameter, you specify the desired row or range of rows in the **Row** parameter, or the **Starting row** and **Ending row** parameters. Similarly, when you select One column or Range of columns from the

Column span parameter, you specify the desired column or range of columns in the **Column** parameter, or the **Starting column** and **Ending column** parameters.

The **Row**, **Column**, **Starting row** or **Starting column** can be specified in six ways:

- First

  For rows, this specifies that the first row of u should be used as the first row of y. When all columns are to be included, this is equivalent to y(1,:) = u(1,:).

  For columns, this specifies that the first column of u should be used as the first column of y. When all rows are to be included, this is equivalent to y(:,1) = u(:,1).

- Index

  For rows, this specifies that the row of u, firstrow, forward-indexed by the **Row index** parameter or the **Starting row index** parameter, should be used as the first row of y. When all columns are to be included, this is equivalent to y(1,:) = u(firstrow,:).

  For columns, this specifies that the column of u, forward-indexed by the **Column index** parameter or the **Starting column index** parameter, firstcol, should be used as the first column of y. When all rows are to be included, this is equivalent to y(:,1) = u(:,firstcol).

- Offset from last

  For rows, this specifies that the row of u offset from row M by the **Row offset** or **Starting row offset** parameter, firstrow, should be used as the first row of y. When all columns are to be included, this is equivalent to y(1,:) = u(M-firstrow,:).

  For columns, this specifies that the column of u offset from column N by the **Column offset** or **Starting column offset** parameter, firstcol, should be used as the first column of y. When all rows are to be included, this is equivalent to y(:,1) = u(:,N-firstcol).

Submatrix

- `Last`

  For rows, this specifies that the last row of u should be used as the only row of y. When all columns are to be included, this is equivalent to `y = u(M,:)`.

  For columns, this specifies that the last column of u should be used as the only column of y. When all rows are to be included, this is equivalent to `y = u(:,N)`.

- `Offset from middle`

  For rows, this specifies that the row of u offset from row M/2 by the **Starting row offset** parameter, `firstrow`, should be used as the first row of y. When all columns are to be included, this is equivalent to `y(1,:) = u(M/2-firstrow,:)`.

  For columns, this specifies that the column of u offset from column N/2 by the **Starting column offset** parameter, `firstcol`, should be used as the first column of y. When all rows are to be included, this is equivalent to `y(:,1) = u(:,N/2-firstcol)`.

- `Middle`

  For rows, this specifies that the middle row of u should be used as the only row of y. When all columns are to be included, this is equivalent to `y = u(M/2,:)`.

  For columns, this specifies that the middle column of u should be used as the only column of y. When all rows are to be included, this is equivalent to `y = u(:,N/2)`.

The **Ending row** or **Ending column** can similarly be specified in five ways:

- `Index`

  For rows, this specifies that the row of u forward-indexed by the **Ending row index** parameter, `lastrow`, should be used as the last row of y. When all columns are to be included, this is equivalent to `y(end,:) = u(lastrow,:)`.

For columns, this specifies that the column of u forward-indexed by the **Ending column index** parameter, lastcol, should be used as the last column of y. When all rows are to be included, this is equivalent to y(:,end) = u(:,lastcol).

- Offset from last

  For rows, this specifies that the row of u offset from row M by the **Ending row offset** parameter, lastrow, should be used as the last row of y. When all columns are to be included, this is equivalent to y(end,:) = u(M-lastrow,:).

  For columns, this specifies that the column of u offset from column N by the **Ending column offset** parameter, lastcol, should be used as the last column of y. When all rows are to be included, this is equivalent to y(:,end) = u(:,N-lastcol).

- Last

  For rows, this specifies that the last row of u should be used as the last row of y. When all columns are to be included, this is equivalent to y(end,:) = u(M,:).

  For columns, this specifies that the last column of u should be used as the last column of y. When all rows are to be included, this is equivalent to y(:,end) = u(:,N).

- Offset from middle

  For rows, this specifies that the row of u offset from row M/2 by the **Ending row offset** parameter, lastrow, should be used as the last row of y. When all columns are to be included, this is equivalent to y(end,:) = u(M/2-lastrow,:).

  For columns, this specifies that the column of u offset from column N/2 by the **Ending column offset** parameter, lastcol, should be used as the last column of y. When all rows are to be included, this is equivalent to y(:,end) = u(:,N/2-lastcol).

- Middle

Submatrix

For rows, this specifies that the middle row of u should be used as the last row of y. When all columns are to be included, this is equivalent to `y(end,:) = u(M/2,:)`.

For columns, this specifies that the middle column of u should be used as the last column of y. When all rows are to be included, this is equivalent to `y(:,end) = u(:,N/2)`.

This block supports Simulink virtual buses.

Examples

To extract the lower-right 3-by-2 submatrix from a 5-by-7 input matrix, enter the following set of parameters:

- **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

The figure below shows the operation for a 5-by-7 matrix with random integer elements, `randint(5,7,10)`.
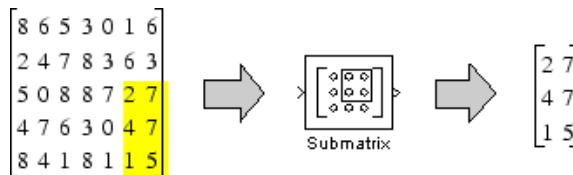
There are often several possible parameter combinations that select the *same* submatrix from the input. For example, instead of specifying Last for **Ending column**, you could select the same submatrix by specifying

- **Ending column** = Index

- **Ending column index** = 7

**Dialog
Box**
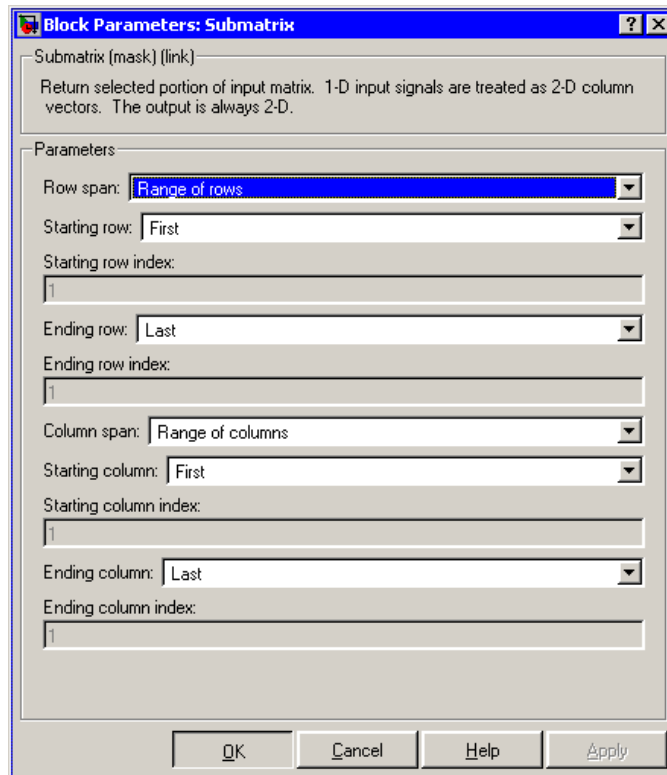
The parameters displayed in the dialog box vary for different menu
combinations. Only some of the parameters listed below are visible
in the dialog box at any one time.

Row span

The range of input rows to be retained in the output. Options are
All rows, One row, or Range of rows.

Submatrix

Row/Starting row
> The input row to be used as the first row of the output. **Row** is enabled when you select `One row` from **Row span**, and **Starting row** when you select `Range of rows` from **Row span**.

Row index/Starting row index
> The index of the input row to be used as the first row of the output. **Row index** is enabled when you select `Index` from Row, and **Starting row index** when you select `Index` from **Starting row**.

Row offset/Starting row offset
> The offset of the input row to be used as the first row of the output. **Row offset** is enabled when you select `Offset from middle` or `Offset from last` from **Row**, and Starting row offset is enabled when you select `Offset from middle` or `Offset from last` from **Starting row**.

Ending row
> The input row to be used as the last row of the output. This parameter is enabled when you select `Range of rows` from **Row span** and you select any option but `Last` from **Starting row**.

Ending row index
> The index of the input row to be used as the last row of the output. This parameter is enabled when you select `Index` from **Ending row**.

Ending row offset
> The offset of the input row to be used as the last row of the output. This parameter is enabled when you select `Offset from middle` or `Offset from last` from **Ending row**.

Column span
> The range of input columns to be retained in the output. Options are `All columns`, `One column`, or `Range of columns`.

Column/Starting column
> The input column to be used as the first column of the output. **Column** is enabled when you select `One column` from **Column**

span, and **Starting column** is enabled when you select Range of columns from **Column span**.

Column index/Starting column index

> The index of the input column to be used as the first column of the output. **Column index** is enabled when you select Index from Column, and **Starting column index** is enabled when you select Index from **Starting column**.

Column offset/Starting column offset

> The offset of the input column to be used as the first column of the output. **Column offset** is enabled when you select Offset from middle or Offset from last from Column. **Starting column offset** is enabled when you select Offset from middle or Offset from last from **Starting column**.

Ending column

> The input column to be used as the last column of the output. This parameter is enabled when you select Range of columns from **Column span** and you select any option but Last from **Starting column**.

Ending column index

> The index of the input column to be used as the last column of the output. This parameter is enabled when you select Index from **Ending column**.

Ending column offset

> The offset of the input column to be used as the last column of the output. This parameter is enabled when you select Offset from middle or Offset from last from **Ending column**.

Submatrix

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Custom data types • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Custom data types • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Reshape	Simulink
Selector	Simulink
Variable Selector	Signal Processing Blockset
reshape	MATLAB

See "Splitting Multichannel Sample-Based Signals into Several Multichannel Signals" on page 1-43 for related information.

SVD Solver

Purpose   Solve AX=B using singular value decomposition

Library   Math Functions / Matrices and Linear Algebra / Linear System Solvers
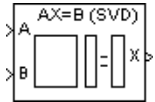
Description   The SVD Solver block solves the linear system AX=B, which can be overdetermined, underdetermined, or exactly determined. The system is solved by applying singular value decomposition (SVD) factorization to the M-by-N matrix, A, at the A port. The input to the B port is the right side M-by-L matrix, B. A length-M 1-D vector input at either port is treated as an M-by-1 matrix.

The output at the x port is the N-by-L matrix, X. X is always sample based, and is chosen to minimize the sum of the squares of the elements of B-AX. When B is a vector, this solution minimizes the vector 2-norm of the residual (B-AX is the residual). When B is a matrix, this solution minimizes the matrix Frobenius norm of the residual. In this case, the columns of X are the solutions to the L corresponding systems $AX_k=B_k$, where B_k is the kth column of B, and X_k is the kth column of X.

X is known as the minimum-norm-residual solution to AX=B. The minimum-norm-residual solution is unique for overdetermined and exactly determined linear systems, but it is not unique for underdetermined linear systems. Thus when the SVD Solver block is applied to an underdetermined system, the output X is chosen such that the number of nonzero entries in X is minimized.

Dialog Box

Block Parameters: SVD Solver

SVD Solver (mask)

Solve AX=B using Singular Value Decomposition. B must have the same number of rows as A. If A is not square, the output is a least squares solution.

[OK]   [Cancel]   [Help]   [Apply]

Supported Data Types

- Double-precision floating point

- Single-precision floating point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Autocorrelation LPC	Signal Processing Blockset
Cholesky Solver	Signal Processing Blockset
LDL Solver	Signal Processing Blockset
Levinson-Durbin	Signal Processing Blockset
LU Inverse	Signal Processing Blockset
Pseudoinverse	Signal Processing Blockset
QR Solver	Signal Processing Blockset
Singular Value Decomposition	Signal Processing Blockset

See "Solving Linear Systems" on page 6-7 for related information.

Time Scope

Purpose   Display signals generated during simulation

Library   Signal Processing Sinks

The Time Scope block is the same as the Scope block in Simulink. To learn how to use the Time Scope block, see the Scope block reference page in the Simulink documentation.

Supported Data Types

Port	Supported Data Types
Input	• Any data type supported by the Scope block

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Scope   Simulink

Purpose   Apply a variable IIR filter to the input.

Library   dspobslib

Description

Note The Time-Varying Direct-Form II Transpose Filter block is still
supported but is likely to be obsoleted in a future release. We strongly
recommend replacing this block with the Digital Filter block.

The Time-Varying Direct-Form II Transpose Filter block is a version of
the Direct-Form II Transpose Filter block whose filter coefficients can
be updated during the simulation. The block applies a direct-form II
transposed IIR filter to the top input (In).
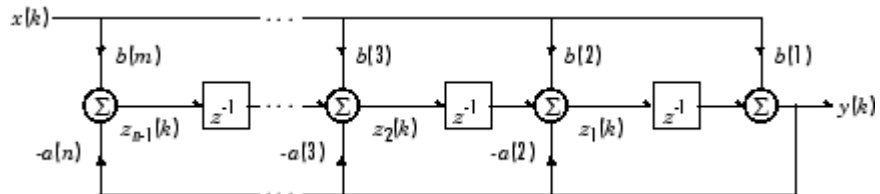
This is a canonical form that has the minimum number of delay
elements. The filter order is max(m,n)-1.

An M-by-N sample-based matrix input is treated as M*N independent
channels, and an M-by-N frame-based matrix input is treated as N
independent channels. In both cases, the block filters each channel
independently over time, and the output has the same size and frame
status as the input.

The block's two lower inputs (Num and Den) specify the filter's transfer
function,

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_1 + b_2 z^{-1} + \ldots + b_{m+1} z^{-(m-1)}}{a_1 + a_2 z^{-1} + \ldots + a_{n+1} z^{-(n-1)}}$$

By default the filter coefficients are normalized by a_1. To prevent normalization by a_1, deselect the **Support non-normalized filters** check box.

Filter Type

The **Filter type** parameter specifies whether the filter is an all-zero (FIR or MA) filter, all-pole (AR) filter, or pole-zero (IIR or ARMA) filter:

- **Pole-zero**

  The block accepts inputs for both the numerator (`Num`) and denominator (`Den`) vectors.

  Input `Num` is a vector of numerator coefficients,

  ```
  [b(1) b(2) ... b(m)]
  ```

  and input `Den` is a vector of denominator coefficients,

  ```
  [a(1) a(2) ... a(n)]
  ```

- **All-zero**

  The block accepts only the numerator vector (`Num`). The denominator of the all-zero filter is 1.

- **All-pole**

  The block accepts only the denominator vector (`Den`). The numerator of the all-pole filter is 1.

For any of these designs, the coefficient vector inputs can change over time to alter the filter's response characteristics during the simulation.

Initial Conditions

In its default form, the filter initializes the internal filter states to zero, which is equivalent to assuming past inputs and outputs are zero. The block also accepts optional nonzero initial conditions for the filter delays. Note that the number of filter states (delay elements) per input channel is

```
max(m,n)-1
```

The **Initial conditions** parameter may take one of four forms:

- Empty matrix

  The empty matrix, [ ], causes a zero (0) initial condition to be applied to all delay elements in each filter channel.

- Scalar

  The scalar value is copied to all delay elements in each filter channel. Note that a value of zero is equivalent to setting the **Initial conditions** parameter to the empty matrix, [ ].

- Vector

  The vector has a length equal to the number of delay elements in each filter channel, max(m,n)-1, and specifies a unique initial condition for each delay element in the filter channel. This vector of initial conditions is applied to each filter channel.

- Matrix

  The matrix specifies a unique initial condition for each delay element, and can specify different initial conditions for each filter channel. The matrix must have the same number of rows as the number of delay elements in the filter, max(m,n)-1, and must have one column per filter channel.

Filter Update Rate

In frame-based operation, the **Filter update rate** parameter determines how frequently the block updates the filter coefficients (i.e., how often it checks the Num and Den inputs). There are two available options:

- **One filter per sample time**

  The block updates the filter coefficients (from inputs Num and Den) for each individual scalar sample in the frame-based input. This

means that each output sample could potentially be computed by a different filter (assuming that Num and Den inputs are updated frequently enough).

- **One filter per frame time**

    The block updates the filter coefficients (from inputs Num and Den) for each new input frame, rather than at each sample in the frame. This means that each output sample in a given frame is a result of an identical filtering process.

Dialog Box
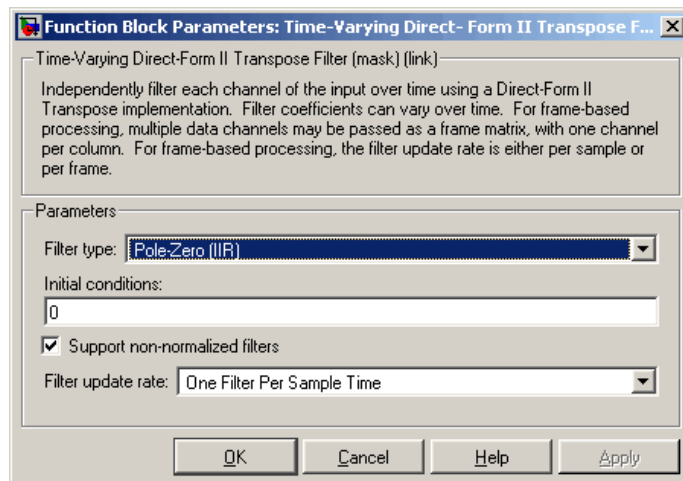
Filter type
> The type of filter to apply: **Pole-Zero (IIR)**, **All-Zero (FIR)**, or **All-Pole (AR)**. The Num and Den input ports are enabled or disabled as appropriate.

Initial conditions
> The filter's initial conditions, a scalar, vector, or matrix.

Support non-normalized filters
> Normalizes the filter by a_1 when selected.

Filter update rate
> The frequency with which the block updates the filter coefficients; once per sample, or once per frame.

References
Oppenheim, A. V. and R. W. Schafer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989.

Proakis, J. and D. Manolakis. *Digital Signal Processing.* 3rd ed. Englewood Cliffs, NJ: Prentice-Hall, 1996.

Time-Varying Lattice Filter

Purpose   Apply a variable lattice filter to the input.

Library   dspobslib

Description
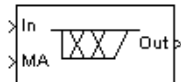
> **Note** The Time-Varying Lattice Filter block is still supported but is likely to be obsoleted in a future release. We strongly recommend replacing this block with the Digital Filter block.

The Time-Varying Lattice Filter block applies a moving average (MA) or autoregressive (AR) lattice filter to the top input (In). The filter reflection coefficients are specified by the vector input to the MA or AR port, and can vary with time.

An M-by-N sample-based matrix input to the In port is treated as M*N independent channels, and an M-by-N frame-based matrix input is treated as N independent channels. In both cases, the block filters each channel independently over time, and the output has the same size and frame status as the input.

Filter Type

The **Filter type** parameter specifies whether the filter is an all-zero (FIR or MA) filter or all-pole (AR) filter.

- **All-zero**

  The block constructs an *n*th order MA filter using the *n* reflection coefficients contained in the vector input to the MA port.

      k = [k(1) k(2) ... k(n)]

- **All-pole**

  The block constructs an *n*th order AR filter using the *n* reflection coefficients contained in the vector input to the AR port.

      k = [k(1) k(2) ... k(n)]

For both designs, the coefficient vector inputs can change over time to alter the filter's response characteristics during the simulation.

Initial Conditions

In its default form, the filter initializes the internal filter states to zero, which is equivalent to assuming past inputs and outputs are zero. The block also accepts optional nonzero initial conditions for the filter delays. Note that the number of filter states (delay elements) per input channel is

```
length(k)
```

The **Initial conditions** parameter may take one of four forms:

- Empty matrix

  The empty matrix, [], causes a zero (0) initial condition to be applied to all delay elements in each filter channel.

- Scalar

  The scalar value is copied to all delay elements in each filter channel. Note that a value of zero is equivalent to setting the **Initial conditions** parameter to the empty matrix.

- Vector

  The vector has a length equal to the number of delay elements in each filter channel, length(k), and specifies a unique initial condition for each delay element in the filter channel. This vector of initial conditions is applied to each filter channel.

- Matrix

  The matrix specifies a unique initial condition for each delay element, and can specify different initial conditions for each filter channel. The matrix must have the same number of rows as the number of delay elements in the filter, length(k), and must have one column per filter channel.

Time-Varying Lattice Filter

Filter Update Rate

In frame-based operation, the **Filter update rate** parameter determines how frequently the block updates the filter coefficients (i.e., how often it checks the MA or AR input). There are two available options:

- **One filter per sample time**

  The block updates the filter coefficients (from input MA or AR) for each individual scalar sample in the framed input. This means that each output sample could potentially be computed by a different filter (assuming that the MA or AR input is updated frequently enough).

- **One filter per frame time**

  The block updates the filter coefficients (from input MA or AR) for each new input frame, rather than at each sample in the frame. This means that each output sample in a given frame is a result of an identical filtering process.

Dialog Box
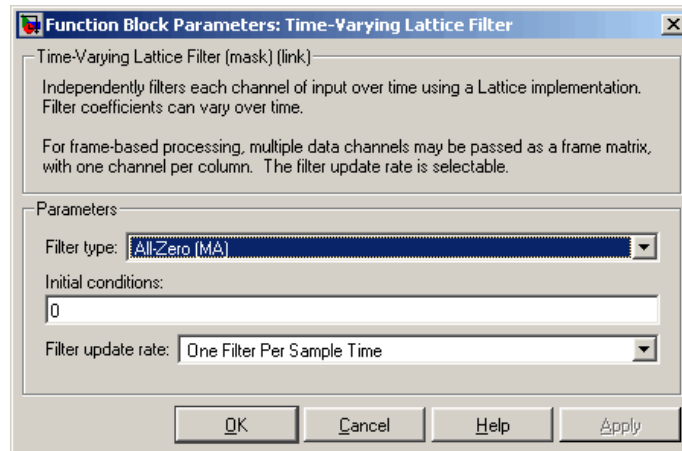
Filter type

The type of filter to apply: MA or AR. The MA or AR input port is enabled or disabled appropriately.

Initial conditions

The filter's initial conditions.

Filter update rate

The frequency with which the block updates the filter coefficients; once per sample, or once per frame.

References   Oppenheim, A. V. and R. W. Schafer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989.

Proakis, J. and D. Manolakis. *Digital Signal Processing*. 3rd ed. Englewood Cliffs, NJ: Prentice-Hall, 1996.

Toeplitz

Purpose   Generate matrix with Toeplitz symmetry

Library   Math Functions / Matrices and Linear Algebra / Matrix Operations

Description
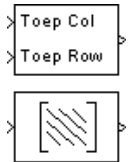
The Toeplitz block generates a Toeplitz matrix from inputs defining the first column and first row. The top input (Col) is a vector containing the values to be placed in the first *column* of the matrix, and the bottom input (Row) is a vector containing the values to be placed in the first *row* of the matrix.

```
y = toeplitz(Col,Row)        % Equivalent MATLAB code
```

The other elements of the matrix obey the relationship

```
y(i,j) = y(i-1,j-1)
```

and the output has dimension [length(Col) length(Row)]. The y(1,1) element is inherited from the Col input. For example, the following inputs

```
Col = [1 2 3 4 5]
Row = [7 7 3 3 2 1 3]
```

produce the Toeplitz matrix

$$
\begin{bmatrix}
1 & 7 & 3 & 3 & 2 & 1 & 3 \\
2 & 1 & 7 & 3 & 3 & 2 & 1 \\
3 & 2 & 1 & 7 & 3 & 3 & 2 \\
4 & 3 & 2 & 1 & 7 & 3 & 3 \\
5 & 4 & 3 & 2 & 1 & 7 & 3
\end{bmatrix}
$$

When both of the inputs are sample based, the output is sample based. Otherwise, the output is frame based.

When you select the **Symmetric** check box, the block generates a symmetric (Hermitian) Toeplitz matrix from a single input, u, defining both the first row and first column of the matrix.

```
y = toeplitz(u)      % Equivalent MATLAB code
```

The output has dimension [length(u) length(u)]. For example, the Toeplitz matrix generated from the input vector [1 2 3 4] is

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

The output has the same frame status as the input.

The Toeplitz block supports real and complex floating-point and fixed-point inputs.

Dialog Box
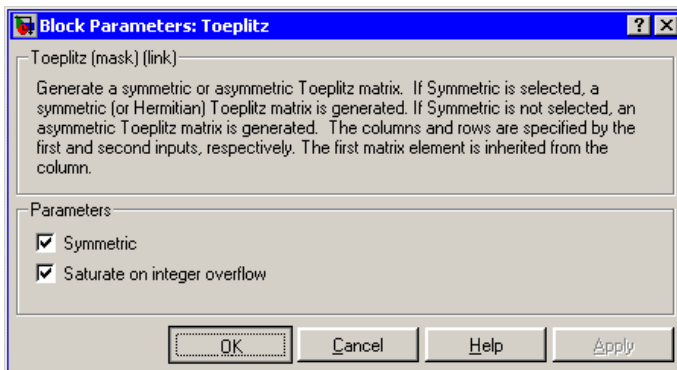
Symmetric

When selected, enables the single-input configuration for symmetric Toeplitz matrix output.

Saturate on integer overflow

When you generate a symmetric Toeplitz matrix with this block, if the input vector is complex, the output is a symmetric Hermitian matrix whose elements satisfy the relationship

$$y(i, j) = \text{conj}(y(j, i))$$

For fixed-point signals the conjugate operation could result in an overflow. When you select this parameter, overflows saturate. This parameter is only visible with the **Symmetric** parameter is selected. This parameter is ignored for floating-point signals.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Custom data types • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers (real signals only)
Toep Col	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Custom data types • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

Port	Supported Data Types
Toep Row	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Custom data types • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Custom data types • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Constant Diagonal Matrix	Signal Processing Blockset
toeplitz	MATLAB

To Multimedia File

Purpose　　　Write video frames and/or audio samples to multimedia file

Library　　　Platform-specific I/O / Windows (WIN32)

Description　The To Multimedia File block writes video frames and/or audio samples to a multimedia (`.avi`) file. Video processing requires the Video and Image Processing Blockset.

> Audio output.avi
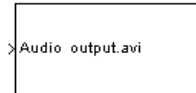
You can also compress the video frames or audio samples by selecting a compression algorithm. You can connect as many of the input ports as you want. Therefore, you can control what type of video and/or audio is sent to the multimedia file.

Note This block supports code generation and is only supported on 32-bit Windows platforms. This block performs best on platforms with DirectX Version 9.0 or later and Windows Media Version 9.0 or later.

Port	Input	Supported Data Types	Supports Complex Values?
R, G, B	Matrix that represents one plane of the RGB video stream. Inputs to the R, G, or B port must have the same dimensions and data type.	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16- 32-bit signed integers • 8-, 16- 32-bit unsigned integers	No
Audio	Vector of audio data	• Double-precision floating point • Single-precision floating point • 16-bit signed integers • 8-bit unsigned integers	No

For the block to display video data properly, double- and single-precision floating-point pixel values must be between 0 and 1. For any other data type, the pixel values must be between the minimum and maximum values supported by their data type.

Use the **Output file name** parameter to specify the name of the multimedia file to which to write. This file is saved in your current directory. To specify a different directory, use the **Browse** button, and then enter the filename, or enter the complete path and filename in the edit box.

Use the **Write** parameter to specify whether the block writes video frames and/or audio samples to the multimedia file. The choices are Video and audio, Video only, or Audio only.

Use the **Video compressor** parameter to specify the type of compression algorithm to use to compress the video data. This compression reduces the size of the multimedia file. Choose None (uncompressed) to save uncompressed video data to the multimedia file. The other items available in this parameter list are the video compression algorithms installed on your system. For information about a specific video compressor, refer to its documentation.

Use the **Audio compressor** parameter to specify the type of compression algorithm to use to compress the audio data. This compression reduces the size of the multimedia file. Choose None (uncompressed) to save uncompressed audio data to the multimedia file. The other items available in this parameter list are the audio compression algorithms installed on your system. For information about a specific audio compressor, refer to its documentation.

Dialog Box
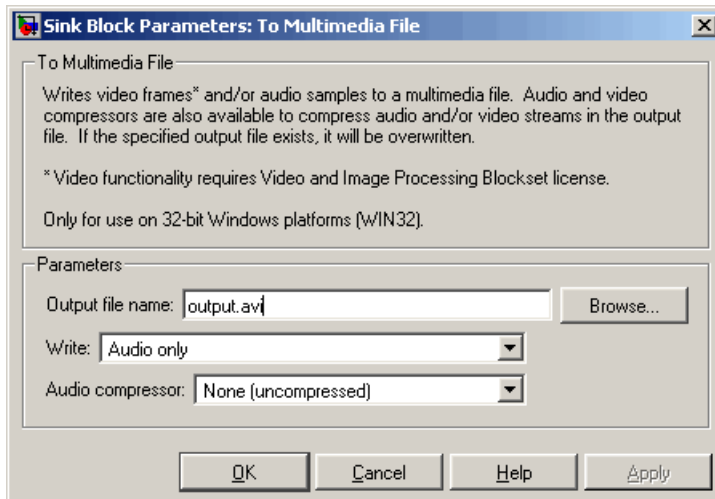
Output file name

Specify the name of the multimedia file to which to write. This file is saved in your current directory. To specify a different directory, use the **Browse** button, and then enter the filename.

Write

Specify whether the block writes video frames and/or audio samples to the multimedia file. The choices are `Video and audio`, `Video only`, or `Audio only`.

Video compressor

Select the type of compression algorithm to use to compress the video data.

Audio compressor

Select the type of compression algorithm to use to compress the audio data.

See Also

From Multimedia File	Signal Processing Blockset
To Wave File	Signal Processing Blockset
Frame Rate Display	Video and Image Processing Blockset
To Video Display	Video and Image Processing Blockset
Video To Workspace	Video and Image Processing Blockset
Video Viewer	Video and Image Processing Blockset

To Wave Device

Purpose   Send audio data to standard audio device in real-time (32-bit Windows operating systems only)

Library   Platform-specific I/O / Windows (WIN32)

Description

The To Wave Device block sends audio data to a standard Windows audio device in real time. It is compatible with most popular Windows hardware, including Sound Blaster cards. (Models that contain both this block and the From Wave Device block require a *duplex-capable* sound card.) The data is sent to the hardware in uncompressed pulse code modulation (PCM) format, and should typically be sampled at one of the standard Windows audio device rates: 8000, 11025, 22050, or 44100 Hz. Some hardware might support other rates in addition to these.

The **Use default audio device** check box allows the block to detect and use the system's default audio hardware. This option should be selected on systems that have a single sound device installed, or when the default sound device on a multiple-device system is the desired target. In cases when the default sound device is *not* the desired output device, clear **Use default audio device**, and set the desired audio device in the **Audio device** parameter, which lists the names of the installed audio device drivers.

The input to the block, *u*, can contain audio data from a mono or stereo signal. A mono signal is represented as either a sample-based scalar or frame-based length-M vector, while a stereo signal is represented as a sample-based length-2 vector or frame-based M-by-2 matrix. When the input data type is uint8, the block conveys the signal samples to the audio device using 8 bits. When the input data type is double, single, or int16, the block conveys the signal samples to the audio device using 16 bits by default. For inputs of data type double and single, you can also set the block to convey the signal samples using 24 bits by selecting the **Enable 24-bit output for double and single precision input signals** check box.

```
sound(u,Fs,bits)     % Equivalent MATLAB code
```

Note that the block does not support `uint16` or `int8` data types. The 16-bit sample width requires more memory but in general yields better fidelity. The amplitude of the input must be in a valid range that depends on the input data type (see the following table). Amplitudes outside the valid range are clipped to the nearest allowable value.

Input Data Type	Valid Input Amplitude Range
double	± 1
single	± 1
int16	-32768 to 32767 (-2^{15} to 2^{15} - 1)
uint8	0 to 255

Buffering

Because the audio device generates real-time audio output, Simulink must maintain a continuous flow of data to the device throughout the simulation. Delays in passing data to the audio hardware can result in hardware errors or distortion of the output. This means that the To Wave Device block must in principle supply data to the audio hardware as quickly as the hardware reads the data. However, the To Wave Device block often *cannot* match the throughput rate of the audio hardware, especially when the simulation is running from within Simulink rather than as generated code. (Simulink execution speed routinely varies during the simulation as the host operating system services other processes.) The block must therefore rely on a buffering strategy to ensure that signal data is accessible to the hardware on demand.

At the start of the simulation, the To Wave Device block writes T_d seconds worth of signal data to the device (hardware) buffer, where you specify T_d in the **Initial output delay** parameter. When this initial data is loaded into the buffer, the audio device begins processing the buffered data, and continues at a constant rate until the buffer empties. You specify the size of the buffer, T_b, in the **Queue duration** parameter. As the audio device reads data from the *front* of the buffer,

the To Wave Device block continues appending inputs to the back of the buffer at the rate they are received.

The following figure shows an audio signal with eight samples per frame. The buffer of the sound board has a five-frame capacity, not fully used at the instant shown. (If the signal sample rate was 8 kHz, for instance, this small buffer could hold approximately 0.005 second of data.)
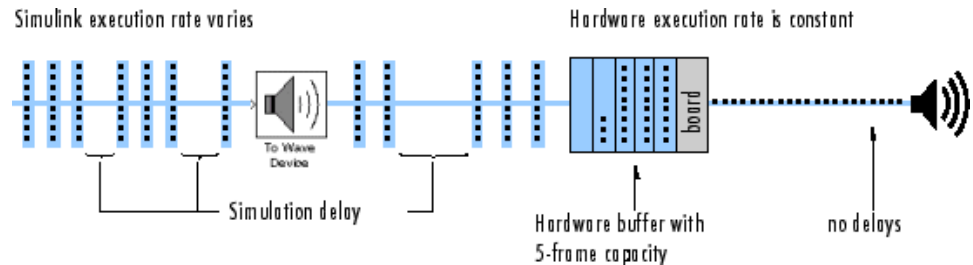
When the simulation throughput rate is higher than the hardware throughput rate, the buffer remains at a constant level throughout the simulation. If necessary, the To Wave Device block buffers inputs until space becomes available in the hardware buffer (that is, data is not thrown away). More typically, the hardware throughput rate is higher than the simulation throughput rate, and the buffer tends to empty over the duration of the simulation.

Under normal operation, an empty buffer indicates that the simulation is finished, and the entire length of the audio signal has been processed. However, when the buffer size is too small in relation to the simulation throughput rate, the buffer might also empty before the entire length of signal is processed. This usually results in a device error or undesired device output.

When the device fails to process the entire signal length because the buffer prematurely empties, you can choose to either increase the buffer size or the simulation throughput rate.

- *Increase the buffer size*. The **Queue duration** parameter specifies the length of signal, T_b (in real-time seconds), to buffer to the audio

device during the simulation. The number of frames buffered is approximately

$$\frac{T_b F_s}{M_o}$$

where F_s is the sample rate of the signal and M_o is the number of samples per frame. The optimal buffer size for a given signal depends on the signal length, the frame size, and the speed of the simulation. The maximum number of frames that can be buffered is 1024.

- *Increase the simulation throughput rate*. Two useful methods for improving simulation throughput rates are increasing the signal frame size and compiling the simulation into native code.

  - Increase frame sizes (and convert sample-based signals to frame-based signals) throughout the model to reduce the amount of block-to-block communication overhead. This can drastically increase throughput rates in many cases. However, larger frame sizes generally result in greater model latency due to initial buffering operations. (Note that increasing the audio signal frame size does not affect the number of samples buffered to the hardware since the **Queue duration** is specified in seconds.)

  - Generate executable code with Real-Time Workshop. Native code runs much faster than Simulink, and should provide rates adequate for real-time audio processing.

Audio problems at startup can often be corrected by entering a larger value for the **Initial output delay** parameter, which allows a greater portion of the signal to be preloaded into the hardware buffer. A value of 0 for the **Initial output delay** parameter specifies the smallest possible initial delay, which is one frame.

More general ways to improve throughput rates include simplifying the model, and running the simulation on a faster PC processor. See the "Simulink" documentation and "Delay and Latency" on page 2-48 for other ideas on improving simulation performance.

To Wave Device

Dialog Box
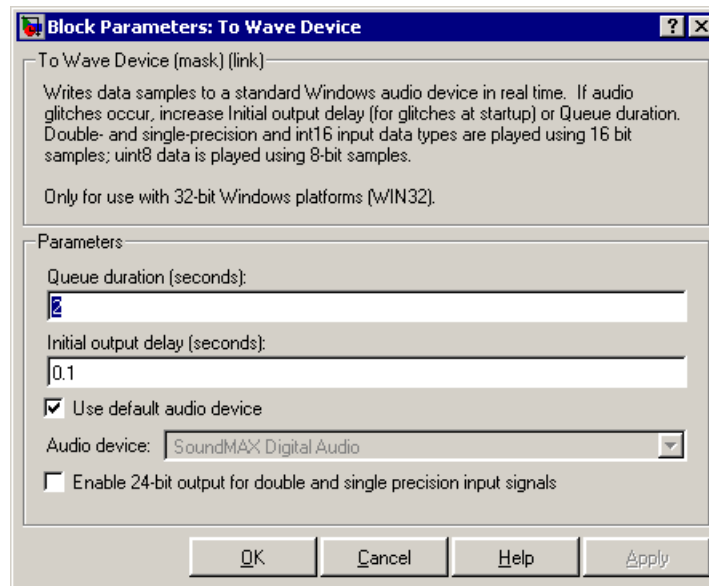
Queue duration (seconds)

The length of signal (in seconds) to buffer to the hardware at the start of the simulation.

Initial output delay (seconds)

The amount of time by which to delay the initial output to the audio device. A value of 0 specifies the smallest possible initial delay, a single frame.

Use default audio device

Directs audio output to the system's default audio device when selected. Clear to enable the **Audio device** parameter and select a device.

Audio device

The name of the audio device to receive the audio output (lists the names of the installed audio device drivers). Select **Use default audio device** when the system has only a single audio card installed.

Enable 24-bit output for double and single precision input signals

Select to output 24-bit data when inputs are double- or single-precision. Otherwise, the block outputs 16-bit data for double- and single-precision inputs.

Supported Data Types

- Double-precision floating point

- Single-precision floating point

- 16-bit signed integer

- 8-bit unsigned integer

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

From Wave Device	Signal Processing Blockset
To Wave File	Signal Processing Blockset
audioplayer	MATLAB
sound	MATLAB

To Wave File

Purpose
Write audio data to file in Microsoft Wave (.wav) format (32-bit Windows operating systems only)

Library
Platform-specific I/O / Windows (WIN32)

Description
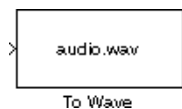
> audio.wav
>
> To Wave

The To Wave File block writes audio data to a Microsoft Wave (.wav) file in the uncompressed pulse code modulation (PCM) format. For compatibility reasons, the sample rate of the discrete-time input signal should typically be one of the standard Windows audio device rates (8000, 11025, 22050, or 44100 Hz), although the block supports arbitrary rates.

The input to the block, *u*, can contain audio data from a mono or stereo signal. A mono signal is represented as either a sample-based scalar or frame-based length-M vector, while a stereo signal is represented as a sample-based length-2 vector or frame-based M-by-2 matrix. The amplitude of the input should be in the range ±1. Values outside this range are clipped to the nearest allowable value.

```
wavwrite(u,Fs,bits,'filename')        % Equivalent MATLAB code
```

The **Sample Width (bits)** parameter specifies the number of bits used to represent the signal samples in the file. These settings are available:

- 8 – allocates 8 bits to each sample, allowing a resolution of 256 levels

- 16 – allocates 16 bits to each sample, allowing a resolution of 65536 levels

- 24 – allocates 24 bits to each sample, allowing a resolution of 16777216 levels

- 32 – allocates 32 bits to each sample, allowing a resolution of 232 levels ranging from -1 to 1

The higher sample width settings require more memory but yield better fidelity for double- and single-precision inputs.

The **File name** parameter can specify an absolute or relative path to the file. You do not need to specify the .wav extension. To reduce the required number of file accesses, the block writes L consecutive samples to the file during each access, where you specify L in the **Minimum number of samples for each write to file** parameter (L ≥M). For L <M, the block instead writes M consecutive samples during each access. Larger values of L result in fewer file accesses, which reduces run-time overhead.

Dialog Box
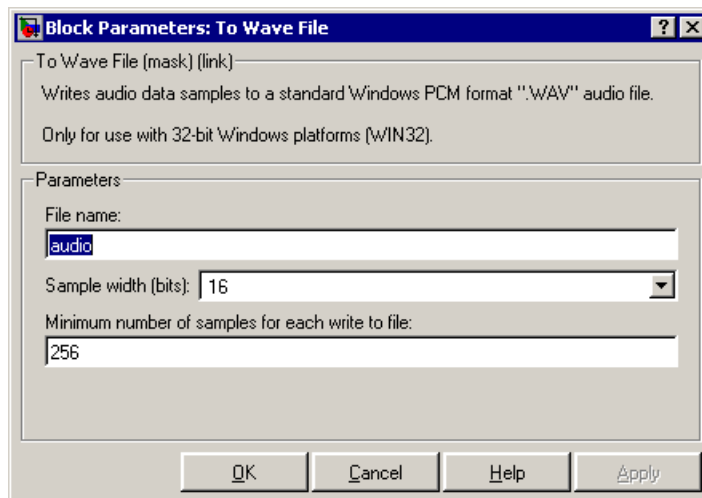
File name

The path and name of the file to write. Paths can be relative or absolute.

Sample width (bits)

The number of bits used to represent each signal sample.

Minimum number of samples for each write to file

The number of consecutive samples to write with each file access, L.

To Wave File

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- 16-bit signed integer
- 8-bit unsigned integer

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

From Wave File	Signal Processing Blockset
To Wave Device	Signal Processing Blockset
To Workspace	Simulink
wavwrite	MATLAB

Purpose　　　Compute matrix transpose

Library　　　Math Functions / Matrices and Linear Algebra / Matrix Operations
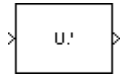
Description　　The Transpose block transposes the M-by-N input matrix to size N-by-M. When you select the **Hermitian** check box, the block performs the Hermitian (complex conjugate) transpose.

```
y = u'     % Equivalent MATLAB code
```

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \end{bmatrix} \quad \xrightarrow{u'} \quad \begin{bmatrix} u_{11}^{*} & u_{21}^{*} \\ u_{12}^{*} & u_{22}^{*} \\ u_{13}^{*} & u_{23}^{*} \end{bmatrix}$$

When you do not select the **Hermitian** check box, the block performs the nonconjugate transpose.

```
y = u.'     % Equivalent MATLAB code
```

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \end{bmatrix} \quad \xrightarrow{u.'} \quad \begin{bmatrix} u_{11} & u_{21} \\ u_{12} & u_{22} \\ u_{13} & u_{23} \end{bmatrix}$$

A length-M 1-D vector input is treated as an M-by-1 matrix. The output is always sample based.

The Transpose block supports real and complex floating-point and fixed-point inputs.

This block also supports Simulink virtual buses.

Transpose

Dialog Box
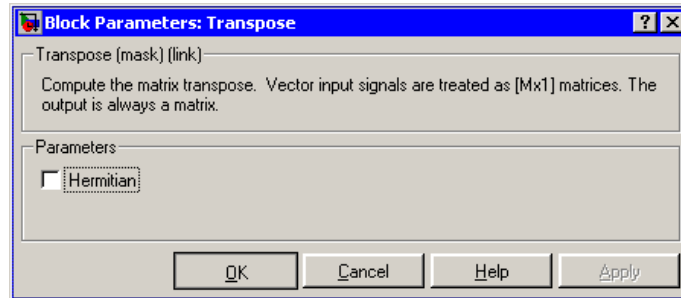
Hermitian

When selected, specifies the complex conjugate transpose. Tunable.

Saturate on integer overflow

This parameter is only visible when the **Hermitian** parameter is selected because overflows can occur when computing the complex conjugate of complex fixed-point signals. When you select this parameter, such overflows saturate. This parameter is ignored for floating-point signals and for real-valued fixed-point signals.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Custom data types
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers (if Hermitian, real signals only)
Output	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Custom data types
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Permute Matrix	Signal Processing Blockset
Reshape	Simulink
Submatrix	Signal Processing Blockset

Triggered Delay Line

Purpose

Buffer sequence of inputs into frame-based output

Library

Signal Management / Buffers

Description
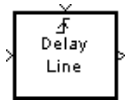
The Triggered Delay Line block acquires a collection of M_o input samples into a frame, where you specify M_o in the **Delay line size** parameter. The block buffers a single sample from input 1 whenever it is triggered by the control signal at input 2 ($\pounds$). When the next triggering event occurs, the newly acquired input sample is appended to the output frame so that the new output overlaps the previous output by M_o-1 samples. Between triggering events the block ignores input 1 and holds the output at its last value.

You specify the triggering event at input 2 in the **Trigger type** pop-up menu:

- `Rising edge` triggers execution of the block when the trigger input rises from a negative value to zero or a positive value, or from zero to a positive value.

- `Falling edge` triggers execution of the block when the trigger input falls from a positive value to zero or a negative value, or from zero to a negative value.

- `Either edge` triggers execution of the block when either a rising or falling edge (as described above) occurs.

The Triggered Delay Line block has zero latency, so the new input appears at the output in the same simulation time step. The output frame period is the same as the input sample period, $T_{fo}=T_{si}$.

Sample-Based Operation

In sample-based operation, the Triggered Delay Line block buffers a sequence of sample-based length-N vector inputs (1-D, row, or column) into a sequence of overlapping sample-based M_o-by-N matrix outputs, where you specify M_o in the **Delay line size** parameter (M_o>1). That is, each input vector becomes a *row* in the sample-based output matrix. When M_o=1, the input is simply passed through to the output, and

retains the same dimension. Sample-based full-dimension matrix inputs are not accepted.

Frame-Based Operation

In frame-based operation, the Triggered Delay Line block rebuffers a sequence of frame-based M_i-by-N matrix inputs into an sequence of overlapping frame-based M_o-by-N matrix outputs, where M_o is the output frame size specified by the **Delay line size** parameter (that is, the number of consecutive samples from the input frame to rebuffer into the output frame). M_o can be greater or less than the input frame size, M_i. Each of the N input channels is rebuffered independently.

Initial Conditions

The Triggered Delay Line block's buffer is initialized to the value specified by the **Initial condition** parameter. The block always outputs this buffer at the first simulation step (t=0). When the block's output is a vector, the **Initial condition** can be a vector of the same size, or a scalar value to be repeated across all elements of the initial output. When the block's output is a matrix, the **Initial condition** can be a matrix of the same size, a vector (of length equal to the number of matrix rows) to be repeated across all columns of the initial output, or a scalar to be repeated across all elements of the initial output.

Examples

In the following three examples, the pulse width of the triggering signal is one sample long, and the delay line size is two. The initial conditions are zero. You can see the difference in results with a rising edge, falling edge, or either edge trigger.

Example 1: Rising Edge Trigger

Open the following model by typing doc_triggereddelay_ref1 at the MATLAB command line.

Triggered Delay Line
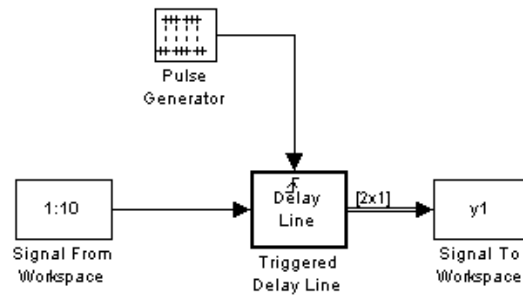
The following shows the input value and Triggered Delay Line output at each time step:
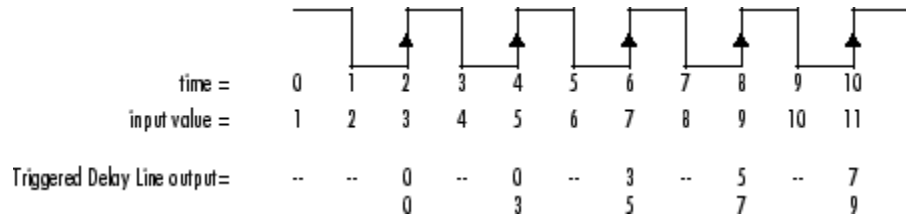
- At t = 0, there are no initial conditions and no trigger.

- At t = 1, there is no trigger.

- At t = 2, there is a rising edge trigger. The output is [0 0] because there were no initial conditions, and the input value at this time, 3, is buffered.

- At t = 3, there is no trigger.

- At t = 4, there is a rising edge trigger. The last buffered value moves into the delay line, which is now [0 3]. Also, the input value at this time, 5, is buffered.

- At t = 5, there is no trigger.

- At t = 6, there is a rising edge trigger. The last buffered value moves into the delay line, which is now [3 5]. Also, the input value at this time, 7, is buffered.

- At t = 7, there is no trigger.

- At t = 8, there is a rising edge trigger. The last buffered value moves into the delay line, which is now [5 7]. Also, the input value at this time, 9, is buffered.

- At t = 9, there is no trigger.

- At t = 10, there is a rising edge trigger. The last buffered value moves into the delay line, which is now [7 9]. Also, the input value at this time, 11, is buffered.

Run the model and look at the output y1 at the MATLAB command line to confirm these results. Remember that when there is no block output, Simulink holds the last value on the line.

Example 2: Falling Edge Trigger

Open the following model by typing doc_triggereddelay_ref2 at the MATLAB command line.
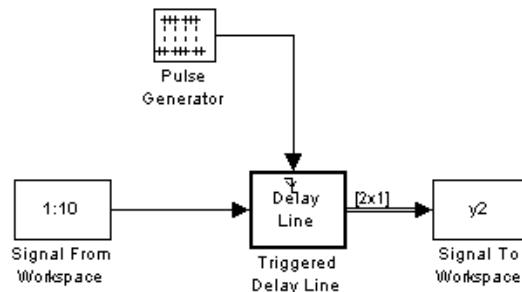
The following shows the input value and Triggered Delay Line output at each time step:

Triggered Delay Line
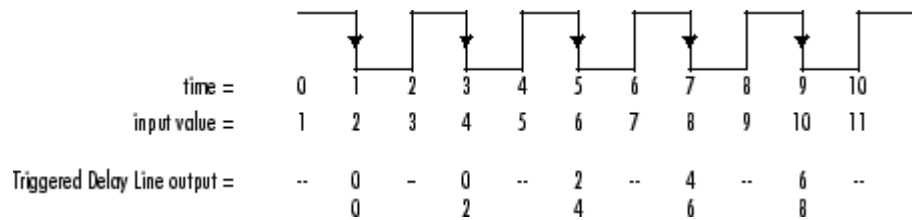
- At t = 0, there are no initial conditions and no trigger.

- At t = 1, there is a falling edge trigger. The output is [0 0] because there were no initial conditions, and the input value at this time, 2, is buffered.

- At t = 2, there is no trigger.

- At t = 3, there is a falling edge trigger. The last buffered value moves into the delay line, which is now [0 2]. Also, the input value at this time, 4, is buffered.

- At t = 4, there is no trigger.

- At t = 5, there is a falling edge trigger. The last buffered value moves into the delay line, which is now [2 4]. Also, the input value at this time, 6, is buffered.

- At t = 6, there is no trigger.

- At t = 7, there is a falling edge trigger. The last buffered value moves into the delay line, which is now [4 6]. Also, the input value at this time, 8, is buffered.

- At t = 8, there is no trigger.

- At t = 9, there is a falling edge trigger. The last buffered value moves into the delay line, which is now [6 8]. Also, the input value at this time, 10, is buffered.

- At t = 10, there is no trigger.

Run the model and look at the output y2 at the MATLAB command line to confirm these results. Remember that when there is no block output, Simulink holds the last value on the line.

Example 3: Either Edge Trigger

Open the following model by typing doc_triggereddelay_ref3 at the MATLAB command line.
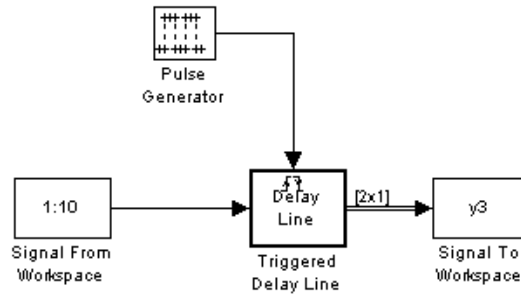
The following shows the input value and Triggered Delay Line output at each time step:
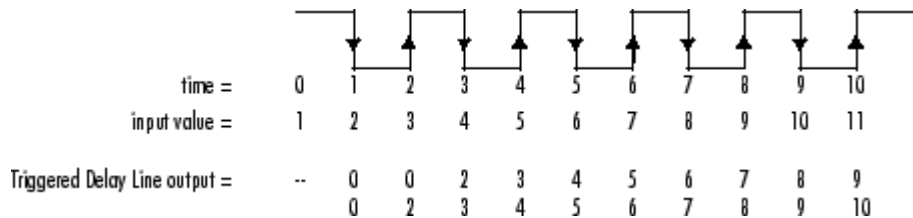
- At t = 0, there are no initial conditions and no trigger.

- At t = 1, there is a falling edge trigger. The output is [0 0], and the input value at this time, 2, is buffered.

- At t = 2, there is a rising edge trigger. The last buffered value moves into the delay line, which is now [0 2]. Also, the input value at this time, 3, is buffered.

- At t = 3, there is a falling edge trigger. The last buffered value moves into the delay line, which is now [2 3]. Also, the input value at this time, 4, is buffered.

- At t = 4, there is a rising edge trigger. The last buffered value moves into the delay line, which is now [3 4]. Also, the input value at this time, 5, is buffered.

- At t = 5, there is a falling edge trigger. The last buffered value moves into the delay line, which is now [4 5]. Also, the input value at this time, 6, is buffered.

- At t = 6, there is a rising edge trigger. The last buffered value moves into the delay line, which is now [5 6]. Also, the input value at this time, 7, is buffered.

- At t = 7, there is a falling edge trigger. The last buffered value moves into the delay line, which is now [6 7]. Also, the input value at this time, 8, is buffered.

- At t = 8, there is a rising edge trigger. The last buffered value moves into the delay line, which is now [7 8]. Also, the input value at this time, 9, is buffered.

- At t = 9, there is a falling edge trigger. The last buffered value moves into the delay line, which is now [8 9]. Also, the input value at this time, 10, is buffered.

- At t = 10, there is a rising edge trigger. The last buffered value moves into the delay line, which is now [9 10]. Also, the input value at this time, 11, is buffered.

Run the model and look at the output y3 at the MATLAB command line to confirm these results. Remember that when there is no block output, Simulink holds the last value on the line.

**Dialog
Box**
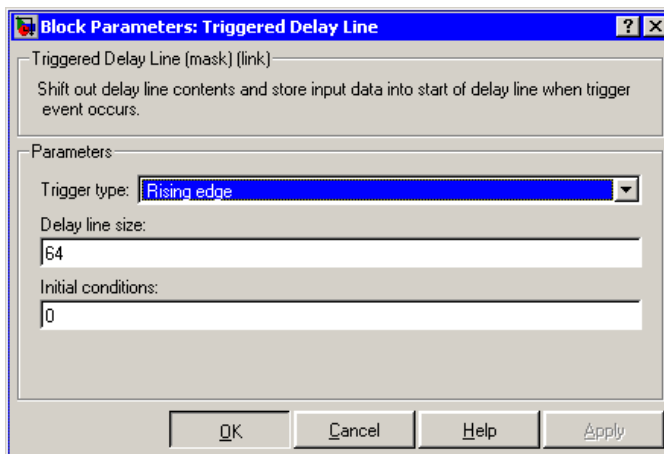
Trigger type

The type of event that triggers the block's execution.

Delay line size

The length of the output frame (number of rows in output matrix), M_o.

Initial condition

The value of the block's initial output, a scalar, vector, or matrix.

**Supported
Data
Types**

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Custom data types
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

Triggered Delay Line

Port	Supported Data Types
Trigger	• Any data type supported by the Trigger block
Output	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Custom data types
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Buffer	Signal Processing Blockset
Delay Line	Signal Processing Blockset
Unbuffer	Signal Processing Blockset

Purpose   Import signal samples from MATLAB workspace when triggered

Library   Signal Operations

Description   The Triggered Signal From Workspace block imports signal samples
from the MATLAB workspace into the Simulink model when triggered
by the control signal at the input port ($\int$). The **Signal** parameter
specifies the name of a MATLAB workspace variable containing the
signal to import, or any valid MATLAB expression defining a matrix
or 3-D array.

When the **Signal** parameter specifies an M-by-N matrix (M≠1), each of
the N columns is treated as a distinct channel. You specify the frame
size in the **Samples per frame** parameter, M_o, and the output when
triggered is an M_o-by-N matrix containing M_o consecutive samples from
each signal channel. For M_o=1, the output is sample based; otherwise
the output is frame based. For convenience, an imported row vector
(M=1) is treated as a single channel, so the output dimension is M_o-by-1.

When the **Signal** parameter specifies an M-by-N-by-P array, the block
generates a single page of the array (an M-by-N matrix) at each trigger
time. The **Samples per frame** parameter must be set to 1, and the
output is always sample based.

Trigger Event

You specify the triggering event at the input port in the **Trigger type**
pop-up menu:

- Rising edge triggers execution of the block when the trigger input
  rises from a negative value to zero or a positive value, or from zero
  to a positive value.

- Falling edge triggers execution of the block when the trigger input
  falls from a positive value to zero or a negative value, or from zero
  to a negative value.

- Either edge triggers execution of the block when either a rising or
  falling edge (as described above) occurs.

Initial and Final Conditions

The **Initial output** parameter specifies the output of the block from the start of the simulation until the first trigger event arrives. Between trigger events, the block holds the output value constant at its most recent value (that is, no linear interpolation takes place). For single-channel signals, the **Initial output** parameter value can be a vector of length M_o or a scalar to repeat across the M_o elements of the initial output frames. For matrix outputs (M_o-by-N or M-by-N), the **Initial output** parameter value can be a vector of length N to repeat across all rows of the initial outputs, or a scalar to repeat across all elements of the initial matrix outputs.

When the block has output all of the available signal samples, it can start again at the beginning of the signal, or simply repeat the final value or generate zeros until the end of the simulation. (The block does not extrapolate the imported signal beyond the last sample.) The **Form output after final data value by** parameter controls this behavior:

- When you specify `Setting To Zero`, the block generates zero-valued outputs for the duration of the simulation after generating the last frame of the signal.

- When you specify `Holding Final Value`, the block repeats the final sample for the duration of the simulation after generating the last frame of the signal.

- When you specify `Cyclic Repetition`, the block repeats the signal from the beginning after generating the last frame. When there are not enough samples at the end of the signal to fill the final frame, the block zero-pads the final frame as necessary to ensure that the output for each cycle is identical (for example, the *i*th frame of one cycle contains the same samples as the *i*th frame of any other cycle).
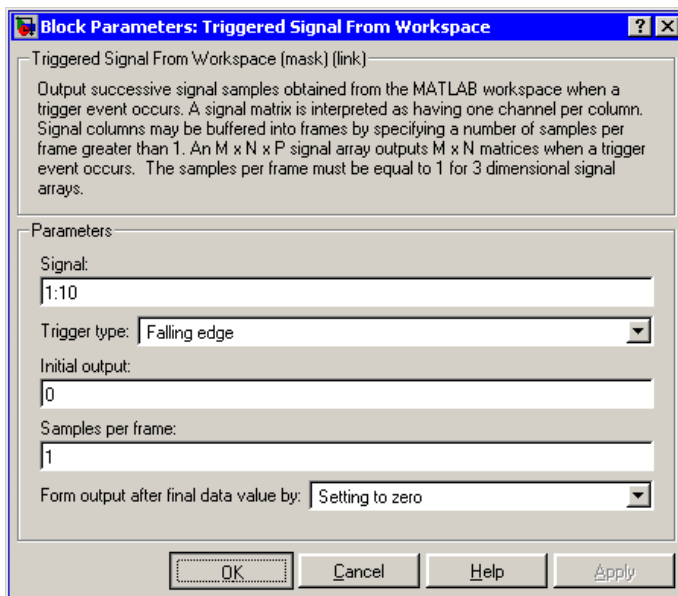
Dialog Box

Signal

    The name of the MATLAB workspace variable from which to import the signal, or a valid MATLAB expression specifying the signal.

Trigger type

    The type of event that triggers the block's execution.

Initial output

    The value to output until the first trigger event is received.

Samples per frame

    The number of samples, M_o, to buffer into each output frame. This value must be 1 when you specify a 3-D array in the **Signal** parameter.

Form output after final data value by

    Specifies the output after all of the specified signal samples have been generated. The block can output zeros for the duration of

Triggered Signal From Workspace

the simulation (`Setting to zero`), repeat the final data sample (`Holding Final Value`) or repeat the entire signal from the beginning (`Cyclic Repetition`).

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point (signed and unsigned)
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

From Wave Device	Signal Processing Blockset
From Wave File	Signal Processing Blockset
Signal To Workspace	Signal Processing Blockset
Signal From Workspace	Signal Processing Blockset
Triggered To Workspace	Signal Processing Blockset

Purpose   Write input sample to MATLAB workspace when triggered

Library   Signal Processing Sinks
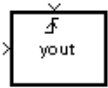
Description   The Triggered To Workspace block creates a matrix or array variable in the MATLAB workspace, where it stores the acquired inputs at the end of a simulation. The block overwrites an existing variable with the same name.

For an M-by-N frame-based input, the block creates an N-column workspace matrix in which each group of M rows represents a single input frame from each of N channels (the most recent frame occupying the last M rows). The maximum size of this workspace variable is limited to P-by-N, where P is the **Maximum number of rows** parameter. (When the simulation progresses long enough for the block to acquire more than P samples, it stores only the most recent P samples.) The **Decimation factor**, D, allows you to store only every Dth input frame.

For an M-by-N sample-based input, the block creates a three-dimensional array in which each M-by-N page represents a single sample from each of M*N channels (the most recent input matrix occupying the last page). The maximum size of this variable is limited to M-by-N-by-P, where P is the **Maximum number of rows** parameter. (When the simulation progresses long enough for the block to acquire more than P inputs, it stores only the last P inputs.) The **Decimation factor**, D, allows you to store only every Dth input matrix.

The block acquires and buffers a single frame from input 1 whenever it is triggered by the control signal at input 2 ($\int$). At all other times, the block ignores input 1. You specify the triggering event at input 2 in the **Trigger type** pop-up menu:

- `Rising edge` triggers execution of the block when the trigger input rises from a negative value to zero or a positive value, or from zero to a positive value.

Triggered To Workspace

- `Falling edge` triggers execution of the block when the trigger input falls from a positive value to zero or a negative value, or from zero to a negative value.

- `Either edge` triggers execution of the block when either a rising or falling edge (as described above) occurs.

To save a record of the sample time corresponding to each sample value, open the Configuration Parameters dialog box. In the **Select** pane, click **Data Import/Export**. In the **Save to workspace** section, select the **Time** check box.

The nontriggered version of this block is the Simulink To Workspace block.

Dialog Box
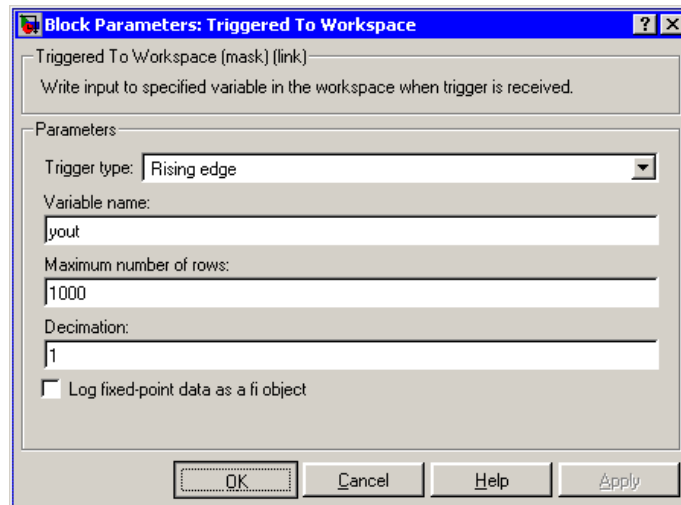
Trigger type
The type of event that triggers the block's execution.

Variable name
The name of the workspace matrix in which to store the data.

Maximum number of rows

    The maximum number of rows (one row per time step) to be saved, P.

Decimation

    The decimation factor, D.

Log fixed-point data as a fi object

    Select to log fixed-point data to the MATLAB workspace as a fi object of the Fixed-Point Toolbox. Otherwise, fixed-point data is logged to the workspace as `double`.

Supported Data Types

Port	Supported Data Types
Input	• Any data type supported by the To Workspace block
Trigger	• Any data type supported by the Trigger block

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Signal From Workspace	Signal Processing Blockset
To Workspace	Simulink

Two-Channel Analysis Subband Filter

Purpose   Decompose signal into a high-frequency subband and a low-frequency subband

Library   Filtering / Multirate Filters

Description   The Two-Channel Analysis Subband Filter block decomposes the input into a high-frequency subband and a low-frequency subband, each with half the bandwidth and half the sample rate of the input.
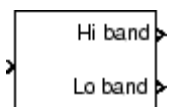
The block filters the input with a pair of highpass and lowpass FIR filters, and then downsamples the results by 2, as illustrated in the following figure.
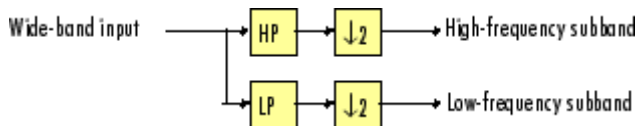
Note that the block implements the FIR filtering and downsampling steps together using a polyphase filter structure, which is more efficient than the straightforward filter-then-decimate algorithm illustrated above. Each subband is the first phase of the respective polyphase filter.

You must provide the vector of filter coefficients for the two filters. Each filter should be a half-band filter that passes the frequency band that the other filter stops. For frame-based inputs, you also need to specify whether the change in the sample rate of the output gets reflected by a change in the frame size, or the frame rate.

Note By connecting many copies of this block, you can implement a multilevel dyadic analysis filter bank. In some cases, it is more efficient to use the Dyadic Analysis Filter Bank block instead. For more information, see "Creating Multilevel Dyadic Analysis Filter Banks" on page 10-1072.

Sections of This Reference Page

Specifying the FIR Filters

You must provide the vector of numerator coefficients for the lowpass and highpass filters in the **Lowpass FIR filter coefficients** and **Highpass FIR filter coefficients** parameters.

For example, to specify a filter with the following transfer function, enter the vector `[b(1) b(2) ... b(m)]`.

$$H(z) = B(z) = b_1 + b_2 z^{-1} + \ldots + b_m z^{-(m-1)}$$

Each filter should be a half-band filter that passes the frequency band that the other filter stops. When you plan to use the Two-Channel Synthesis Subband Filter block to reconstruct the input to this block, you will need to design perfect reconstruction filters to use in the synthesis subband filter.

The best way to design perfect reconstruction filters is to use the `wfilters` function in the Wavelet Toolbox to design both the filters

both in this block and in the Two-Channel Synthesis Subband Filter block. You can also use functions from the Filter Design Toolbox and Signal Processing Toolbox. To learn how to design your own perfect reconstruction filters, see "References" on page 10-1082.

The block initializes all filter states to zero.

Sample-Based Operation

- "Valid Sample-Based Inputs" on page 10-1070
- "Sample-Based Outputs" on page 10-1070

Valid Sample-Based Inputs

The block accepts all M-by-N sample-based matrix inputs. The block treats such inputs as $M \cdot N$ independent channels, and decomposes each channel over time.

Sample-Based Outputs

Given a sample-based M-by-N input, the block outputs two M-by-N sample-based matrices whose sample rates are half the input sample rate. Each output matrix element is the high- or low-frequency subband output of the corresponding input matrix element. Depending on the Simulink configuration parameters, some sample-based outputs can have one sample of latency, as described in "Latency" on page 10-1071.

Frame-Based Operation

- "Valid Frame-Based Inputs" on page 10-1070
- "Frame-Based Outputs" on page 10-1071

Valid Frame-Based Inputs

The block accepts M-by-N frame-based matrix inputs where M is a multiple of two. The block treats such inputs as N independent channels, and decomposes each channel over time.

Frame-Based Outputs

Given a valid frame-based input, the block outputs two frame-based matrices. Each output column is the high- or low-frequency subband of the corresponding input column.

The sample rate of the outputs are half that of the input. The **Framing** parameter sets whether the block halves the sample rate by halving the output frame size, or halving the output frame rate:

- `Maintain input frame size` – The input and output frame *sizes* are the same, but the frame *rate* of the outputs are half that of the input. So, the overall sample rate of the output is half that of the input. This setting causes the block to have one frame of latency, as described in "Latency" on page 10-1071.

- `Maintain input frame rate` – The input and output frame *rates* are the same, but the frame *size* of the outputs are half that of the input (the input frame size must be a multiple of two). So, the overall sample rate of the output is half that of the input.

Latency

In some cases, the block has nonzero tasking latency, which means that there is a constant delay between the time that the block receives an input, and produces the corresponding output, as summarized below and in the following table:

- For sample-based inputs, there are cases where the block exhibits *one-sample latency*. In such cases, when the block receives the nth input sample, it produces the outputs corresponding to the n-*1*th input sample. When the block receives the first input sample, the block outputs an initial value of zero in each output channel.

- For frame-based inputs, there are cases where the block exhibits *one-frame latency*. In such cases, when the block receives the nth input frame, it produces the outputs corresponding to the n-*1*th input frame. When the block receives the first input frame, the block outputs a frame of zeros.

Two-Channel Analysis Subband Filter

> **Note** For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" on page 2-56 and "Models with Multiple Sample Rates" in the Real-Time Workshop User's Guide documentation.

Amount of Block Latency for All Possible Block Settings

Input	Latency	No Latency
Sample based	One sample of latency when the **Tasking mode for periodic sample times** parameter is set to MultiTasking or Auto in the **Solver** pane of the Configuration Parameters dialog box. The first output sample of each channel is always 0.	The **Tasking mode for periodic sample times** parameter is set to SingleTasking in the **Solver** pane of the Configuration Parameters dialog box.
Frame based	One frame of latency when the **Framing** parameter is set to Maintain input frame size. The first output frame is always all zeros.	The **Framing** parameter is set to Maintain input frame rate.

Creating Multilevel Dyadic Analysis Filter Banks

The Two-Channel Analysis Subband Filter block is the basic unit of a dyadic analysis filter bank. You can connect several of these blocks to implement an n-level filter bank, as illustrated in the following figure. For a review of dyadic analysis filter banks, see the Dyadic Analysis Filter Bank block reference page.

When you create a filter bank by connecting multiple copies of this block, the output values of the filter bank differ depending on whether there is latency. See the previous table,

For instance, for frame-based inputs, the filter bank output values differ depending on whether you set the **Framing** parameter to Maintain

input frame rate (no latency), or Maintain input frame size (one frame of latency for every block). Though the output values differ, both sets of values are valid; the difference arises from changes in latency.

In some cases, rather than connecting several Two-Channel Analysis Subband Filter blocks, it is faster and requires less memory to use the Dyadic Analysis Filter Bank block. In particular, use the Dyadic Analysis Filter Bank block when you want to decompose a frame-based signal with frame size a multiple of 2^n into $n+1$ or 2^n subbands. In all other cases, use Two-Channel Analysis Subband Filter blocks to implement your filter banks.
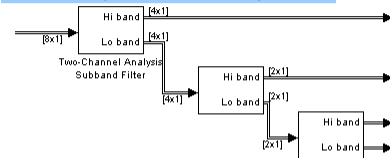
3-Level Dyadic Analysis Filter Banks

The Dyadic Analysis Filter Bank block allows you to specify the filter bank filters by providing vectors of filter coefficients, just as this block does. The Dyadic Analysis Filter Bank block provides an additional option of using wavelet-based filters that the block designs by using a wavelet you specify.

Fixed-Point Data Types

The Two-Channel Analysis Subband Filter block is comprised of two FIR Decimation blocks as shown in the following diagram.

Two-Channel Analysis Subband Filter
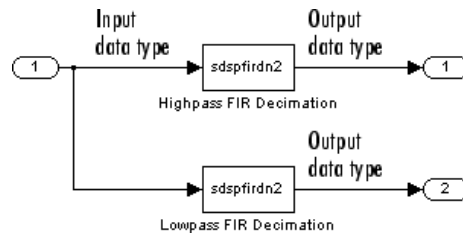
For fixed-point signals, you can set the coefficient, product output, accumulator, and output data types of the FIR Decimation blocks as discussed in "Dialog Box" on page 10-1075. For a diagram showing the usage of these data types, refer to the FIR Decimation block reference page.

Examples   See the following Signal Processing Blockset demos, which use the Two-Channel Analysis Subband Filter block:

- Multilevel PR filter bank

- Denoising

- Wavelet transmultiplexer (WTM)

Note By default, the demos open the versions using the Two-Channel Analysis Subband Filter block. You can also see the version of the demos that use the Dyadic Analysis Filter Bank block by clicking the **Frame-Based Demo** button in the demos.

**Dialog
Box**

The **Main** pane of the Two-Channel Analysis Subband Filter block
dialog appears as follows:
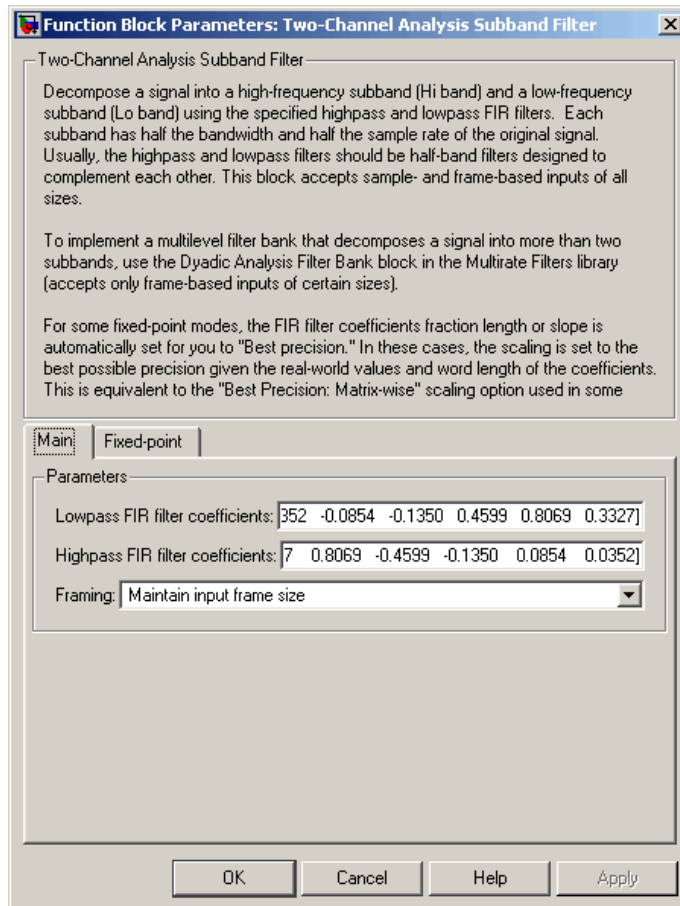
Lowpass FIR filter coefficients

Specify a vector of lowpass FIR filter coefficients, in descending
powers of z. The lowpass filter should be a half-band filter that
passes the frequency band stopped by the filter specified in the
Highpass FIR filter coefficients parameter. The default

values of this parameter specify a filter based on a 3rd-order Daubechies wavelet. When you use the Two-Channel Synthesis Subband Filter block to reconstruct the input to this block, you need to design perfect reconstruction filters to use in the synthesis subband filter. For more information, see "Specifying the FIR Filters" on page 10-1069.

Highpass FIR filter coefficients

Specify a vector of highpass FIR filter coefficients, in descending powers of z. The highpass filter should be a half-band filter that passes the frequency band stopped by the filter specified in the **Lowpass FIR filter coefficients** parameter. The default values of this parameter specify a filter based on a 3rd-order Daubechies wavelet. When you use the Two-Channel Synthesis Subband Filter block to reconstruct the input to this block, you need to design perfect reconstruction filters to use in the synthesis subband filter. For more information, see "Specifying the FIR Filters" on page 10-1069.

Framing

Specify the method by which to implement the decimation for frame-based inputs:

Select `Maintain input frame size` to halve the output frame rate

Select `Maintain input frame rate` to halve the output frame size

For more information, see "Frame-Based Operation" on page 10-1070. Some settings of this parameter causes the block to have nonzero latency, as described in "Latency" on page 10-1071.

The **Fixed-point** pane of the Two-Channel Analysis Subband Filter block dialog appears as follows:

Rounding mode

Select the rounding mode for fixed-point operations. The filter coefficients do not obey this parameter; they always round to Nearest.

Overflow mode

Select the overflow mode for fixed-point operations. The filter coefficients do not obey this parameter; they are always saturated.

Coefficients

Choose how you will specify the word length and the fraction length of the FIR filter coefficients:

When you select `Same word length as input`, the word length of the filter coefficients will match that of the input to the block. In this mode, the fraction length of the coefficients is automatically set to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.

When you select `Specify word length`, you are able to enter the word length of the coefficients, in bits. In this mode, the fraction length of the coefficients is automatically set to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.

When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the coefficients, in bits.

When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the coefficients. This block requires power-of-two slope and a bias of zero.

The filter coefficients do not obey the **Rounding mode** and the **Overflow mode** parameters; they are always saturated and rounded to `Nearest`.

Product output

Use this parameter to specify how you would like to designate the product output word and fraction lengths. Refer to "Fixed-Point Data Types" on page 10-430 of the FIR Decimation reference page and "Multiplication Data Types" on page 8-16 for illustrations depicting the use of the product output data type in the FIR Decimation blocks of this block:

When you select `Inherit via internal rule`, the product output word length and fraction length are automatically set according to the following equations:

$$ideal\ product\ output\ word\ length\ =$$
$$input\ word\ length + FIR\ coefficients\ word\ length$$

$$ideal\ product\ output\ fraction\ length\ =$$
$$input\ fraction\ length + FIR\ coefficients\ fraction\ length$$

Note The actual product output word length may be equal to or greater than the calculated ideal product output word length, depending on the settings on the **Hardware Implementation** pane of the Configuration Parameters dialog box.

When you select `Same as input`, these characteristics will match those of the input to the block.

When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the product output, in bits.

When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the product output. This block requires power-of-two slope and a bias of zero.

Accumulator

As depicted above, inputs to the accumulator are cast to the accumulator data type. The output of the adder remains in the accumulator data type as each element of the input is added to it. Use this parameter to specify how you would like to designate this accumulator word and fraction lengths.

You also use this parameter to specify the accumulator word and fraction lengths resulting from a complex-complex multiplication in the FIR Decimation blocks in this block. Refer to "Multiplication Data Types" on page 8-16 for more information:

When you select `Inherit via internal rule`, the accumulator word length and fraction length are automatically set according to the following equations:

$$\begin{aligned} ideal\ accumulator\ word\ length\ &=\ ideal\ product\ output\ word\ length \\ &+\ \mathrm{floor}(\log 2(number\ of\ accumulations)) + 1 \end{aligned}$$

$$\begin{aligned} ideal\ accumulator\ fraction\ length\ &= \\ ideal\ product\ output\ fraction\ length \end{aligned}$$

where the number of accumulations is given by

$$((number\ of\ coefficients\,/\,decimation\ factor) - 1)$$

if either the coefficients or inputs are real

$$number\ of\ coefficients\,/\,decimation\ factor$$

if both the coefficients and inputs are complex

Note The actual accumulator word length may be equal to or greater than the calculated ideal product output word length, depending on the settings on the **Hardware Implementation** pane of the Configuration Parameters dialog box.

When you select `Same as product output`, these characteristics will match those of the product output

When you select `Same as input`, these characteristics will match those of the input to the block.

When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the accumulator, in bits.

When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the accumulator. This block requires power-of-two slope and a bias of zero.

Output

Choose how you will specify the output word length and fraction length of the FIR Decimation blocks, as well as of the final overall filter output:

When you select `Same as accumulator`, these characteristics will match those of the accumulator.

A special case occurs when `Inherit via internal rule` is specified for **Accumulator**, and block inputs and coefficients are complex. In that case, the output word length will be one less than the accumulator word length.

When you select `Same as product output`, these characteristics will match those of the product output

When you select `Same as input`, these characteristics will match those of the input to the block.

When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the output, in bits.

When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the output. This block requires power-of-two slope and a bias of zero.

Two-Channel Analysis Subband Filter

References Fliege, N. J. *Multirate Digital Signal Processing: Multirate Systems, Filter Banks, Wavelets*. West Sussex, England: John Wiley & Sons, 1994.

Strang, G. and T. Nguyen. *Wavelets and Filter Banks*. Wellesley, MA: Wellesley-Cambridge Press, 1996.

Vaidyanathan, P. P. *Multirate Systems and Filter Banks*. Englewood Cliffs, NJ: Prentice Hall, 1993.

Supported Data Types
- Double-precision floating point
- Single-precision floating point
- Fixed point (signed only)
- 8-, 16-, and 32-bit signed integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Dyadic Analysis Filter Bank	Signal Processing Blockset
FIR Decimation	Signal Processing Blockset
Two-Channel Synthesis Subband Filter	Signal Processing Blockset
fir1	Signal Processing Toolbox
fir2	Signal Processing Toolbox
firls	Signal Processing Toolbox
wfilters	Wavelet Toolbox

For related information, see "Multirate Filters" on page 3-66.

Purpose   Reconstruct signal from a high-frequency subband and a low-frequency subband

Library   Filtering / Multirate Filters

Description

> Hi band
>
> Lo band

The Two-Channel Synthesis Subband Filter block reconstructs a signal from its high-frequency subband and low-frequency subband, each with half the bandwidth and half the sample rate of the original signal. Use this block to reconstruct signals decomposed by the Two-Channel Analysis Subband Filter block.

The block upsamples the high- and low-frequency subbands by 2, and then filters the results with a pair of highpass and lowpass FIR filters, as illustrated in the following figure.
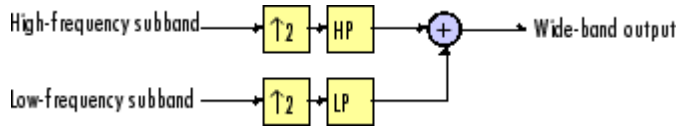
Note that the block implements the FIR filtering and downsampling steps together using a polyphase filter structure, which is more efficient than the straightforward interpolate-then-filter algorithm illustrated above.

You must provide the vector of filter coefficients for the two filters. Each filter should be a half-band filter that passes the frequency band that the other filter stops. To use this block to reconstruct the output of a Two-Channel Analysis Subband Filter block, the filters in this block *must* be designed to perfectly reconstruct the outputs of the analysis filters.

Two-Channel Synthesis Subband Filter

> **Note** By connecting many copies of this block, you can implement a multilevel dyadic synthesis filter bank. In some cases, it is more efficient to use the Dyadic Synthesis Filter Bank block instead. For more information, see "Creating Multilevel Dyadic Synthesis Filter Banks" on page 10-1088.

Sections of This Reference Page

Specifying the FIR Filters

You must provide the vector of numerator coefficients for the lowpass and highpass filters in the **Lowpass FIR filter coefficients** and **Highpass FIR filter coefficients** parameters.

For example, to specify a filter with the following transfer function, enter the vector `[b(1) b(2) ... b(m)]`.

$$H(z) = B(z) = b_1 + b_2 z^{-1} + \ldots + b_m z^{-(m-1)}$$

Each filter should be a half-band filter that passes the frequency band that the other filter stops. To use this block to reconstruct the output of a Two-Channel Analysis Subband Filter block, the filters in this block must be designed to perfectly reconstruct the outputs of the analysis filters.

The best way to design perfect reconstruction filters is to use the `wfilters` function in the Wavelet Toolbox for the filters in both this block *and* in the corresponding Two-Channel Analysis Subband Filter block. You can also use functions from the Filter Design Toolbox and Signal Processing Toolbox. To learn how to design your own perfect reconstruction filters, see "References" on page 10-1098.

The block initializes all filter states to zero.

Sample-Based Operation

- "Valid Sample-Based Inputs" on page 10-1085
- "Sample-Based Outputs" on page 10-1085

Valid Sample-Based Inputs

The block accepts any two M-by-N sample-based matrices with the same sample rates. The block treats each M-by-N matrix as MxN independent subbands, where MxN is the product of the matrix dimensions. Each matrix element is the high- or low-frequency subband of the corresponding channel in the output matrix. The input to the topmost input port should contain the high-frequency subbands.

Sample-Based Outputs

Given valid sample-based inputs, the block outputs one sample-based matrix with the same dimensions as the inputs. The output sample rate is twice that of the input. Each element of the output is a single channel, reconstructed from the corresponding elements in each input matrix. Depending on the Simulink configuration parameters, some sample-based outputs can have one sample of latency, as described in "Latency" on page 10-1086.

Two-Channel Synthesis Subband Filter

Frame-Based Operation

- "Valid Frame-Based Inputs" on page 10-1086
- "Frame-Based Outputs" on page 10-1086

Valid Frame-Based Inputs

The block accepts any two M-by-N frame-based matrices with the same frame rates. The block treats each input column as the high- or low-frequency subbands of the corresponding output channel. The input to the topmost input port should contain the high-frequency subbands.

Frame-Based Outputs

Given valid frame-based inputs, the block outputs a frame-based matrix. Each output column is a single channel, reconstructed from the corresponding columns in each input matrix.

The sample rate of the output is twice that of the input. The **Framing** parameter sets whether the block doubles the sample rate by doubling the output frame size, or doubling the output frame rate:

- Maintain input frame size – The input and output frame *sizes* are the same, but the frame *rate* of the output is twice that of the input. So, the overall sample rate of the output is twice that of the input. This setting causes the block to have one frame of latency, as described in "Latency" on page 10-1071.

- Maintain input frame rate – The input and output frame *rates* are the same, but the frame *size* of the output is twice that of the input. So, the overall sample rate of the output is twice that of the input.

Latency

In some cases, the block has nonzero tasking latency, which means that there is a constant delay between the time that the block receives an input, and produces the corresponding output, as summarized below and in the following table:

- For sample-based inputs, there are cases where the block exhibits *one-sample latency*. In such cases, when the block receives the nth input sample, it produces the outputs corresponding to the $n\text{-}1$th input sample. When the block receives the first input sample, the block outputs an initial value of zero in each output channel.

- For frame-based inputs, there are cases where the block exhibits *one-frame latency*. In such cases, when the block receives the nth input frame, it produces the outputs corresponding to the $n\text{-}1$th input frame. When the block receives the first input frame, the block outputs a frame of zeros.

Note For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" on page 2-56 and "Models with Multiple Sample Rates" in the Real-Time Workshop User's Guide documentation.

Amount of Block Latency for All Possible Block Settings

Input	Latency	No Latency
Sample based	One sample of latency when the **Tasking mode for periodic sample times** parameter is set to `MultiTasking` or `Auto` in the **Solver** pane of the Configuration Parameters dialog box. The first output sample of each channel is always `0`.	The **Tasking mode for periodic sample times** parameter is set to `SingleTasking` in the **Solver** pane of the Configuration Parameters dialog box.
Frame based	One frame of latency when the **Framing** parameter is set to `Maintain input frame size`. The first output frame is always all zeros.	The **Framing** parameter is set to `Maintain input frame rate`.

Two-Channel Synthesis Subband Filter

Creating Multilevel Dyadic Synthesis Filter Banks

The Two-Channel Synthesis Subband Filter block is the basic unit of a dyadic synthesis filter bank. You can connect several of these blocks to implement an n-level filter bank, as illustrated in the following figure. For a review of dyadic synthesis filter banks, see the Dyadic Synthesis Filter Bank block reference page.

When you create a filter bank by connecting multiple copies of this block, the output values of the filter bank differ depending on whether there is latency. See the previous table, Amount of Block Latency for All Possible Block Settings on page 10-1087.

For instance, for frame-based inputs, the filter bank output values differ depending on whether you set the **Framing** parameter to `Maintain input frame rate` (no latency), or `Maintain input frame size` (one frame of latency for every block). Though the output values differ, both sets of values are valid; the difference arises from changes in latency.

In some cases, rather than connecting several Two-Channel Synthesis Subband Filter blocks, it is faster and requires less memory to use the Dyadic Synthesis Filter Bank block. In particular, use the Dyadic Synthesis Filter Bank block to reconstruct a frame-based signal (with frame size a multiple of 2^n) from 2^n or $n+1$ subbands whose properties match those of the Dyadic Analysis Filter Bank block's outputs. These properties are described in the Dyadic Analysis Filter Bank reference page.

3-Level Dyadic Synthesis Filter Banks

The Dyadic Synthesis Filter Bank block allows you to specify the filter bank filters by providing vectors of filter coefficients, just as this block does. The Dyadic Synthesis Filter Bank block provides an additional option of using wavelet-based filters that the block designs by using a wavelet you specify.

Fixed-Point Data Types

The Two-Channel Synthesis Subband Filter block is comprised of two FIR Interpolation blocks as shown in the following diagram.
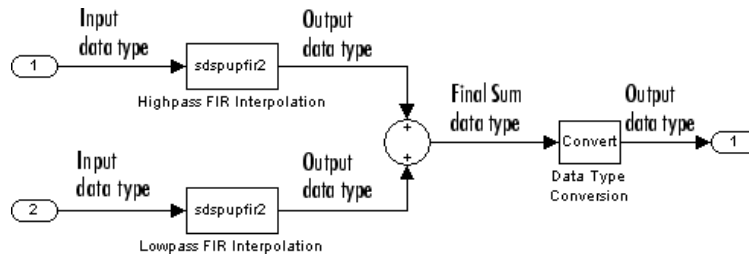
For fixed-point signals, you can set the coefficient, product output, accumulator, and output data types used in the FIR Interpolation blocks as discussed in "Dialog Box" on page 10-1091 below. For a diagram showing the usage of these data types within the FIR blocks, refer to the FIR Interpolation block reference page.

Two-Channel Synthesis Subband Filter

In addition, the inputs to the Sum block in the diagram above are accumulated using the accumulator data type. The output of the Sum block is then cast from the accumulator data type to the output data type. Therefore the output of the Two-Channel Synthesis Subband Filter block is in the output data type. You also set these data types in the block dialog as discussed in "Dialog Box" on page 10-1091 below.

Examples   See the following Signal Processing Blockset demos, which use the Two-Channel Synthesis Subband Filter block:

- Multilevel PR filter bank

- Denoising

- Wavelet transmultiplexer (WTM)

Note By default, the demos open the versions using the Two-Channel Synthesis Subband Filter block. You can also see the version of the demos that use the Dyadic Synthesis Filter Bank block by clicking the **Frame-Based Demo** button in the demos.

**Dialog
Box**

The **Main** pane of the Two-Channel Synthesis Subband Filter block dialog appears as follows:
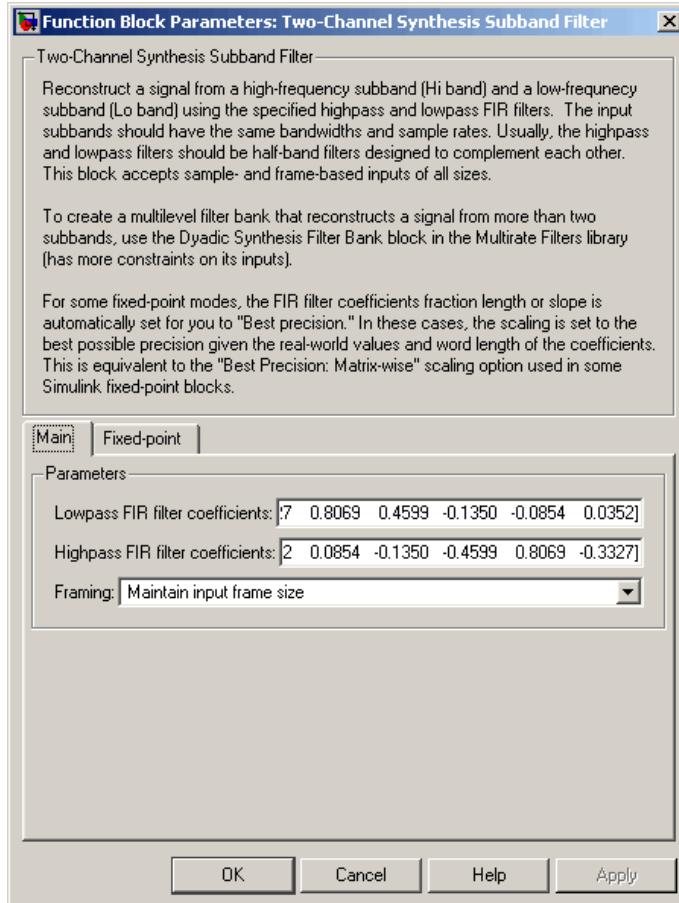
Lowpass FIR filter coefficients

A vector of lowpass FIR filter coefficients, in descending powers of z. The lowpass filter should be a half-band filter that passes the frequency band stopped by the filter specified in the **Highpass FIR filter coefficients** parameter. To use

this block to reconstruct the output of a Two-Channel Analysis Subband Filter block, you must design the filters in this block to perfectly reconstruct the outputs of the analysis filters. For more information, see "Specifying the FIR Filters" on page 10-1084.

Highpass FIR filter coefficients

A vector of highpass FIR filter coefficients, in descending powers of z. The highpass filter should be a half-band filter that passes the frequency band stopped by the filter specified in the **Lowpass FIR filter coefficients** parameter. To use this block to reconstruct the output of a Two-Channel Analysis Subband Filter block, you must design the filters in this block to perfectly reconstruct the outputs of the analysis filters. For more information, see "Specifying the FIR Filters" on page 10-1084.

Framing

Select the method by which to implement the interpolation for frame-based inputs:

Select `Maintain input frame size` to double the output frame rate

Select `Maintain input frame rate` to double the output frame size

For more information, see "Frame-Based Operation" on page 10-1070. Some settings of this parameter causes the block to have nonzero latency, as described in "Latency" on page 10-1071.

The **Fixed-point** pane of the Two-Channel Synthesis Subband Filter block dialog appears as follows:

Round mode

Select the rounding mode for fixed-point operations. The filter coefficients do not obey this parameter; they always round to Nearest.

Overflow mode

Select the overflow mode for fixed-point operations. The filter coefficients do not obey this parameter; they are always saturated.

Coefficients

Choose how you will specify the word length and the fraction length of the FIR filter coefficients:

When you select `Same word length as input`, the word length of the filter coefficients will match that of the input to the block. In this mode, the fraction length of the coefficients is automatically set to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.

When you select `Specify word length`, you are able to enter the word length of the coefficients, in bits. In this mode, the fraction length of the coefficients is automatically set to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.

When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the coefficients, in bits.

When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the coefficients. This block requires power-of-two slope and a bias of zero.

The filter coefficients do not obey the **Rounding mode** and the **Overflow mode** parameters; they are always saturated and rounded to `Nearest`.

Product output

Use this parameter to specify how you would like to designate the product output word and fraction lengths. Refer to "Fixed-Point Data Types" on page 10-446 of the FIR Interpolation reference page and "Multiplication Data Types" on page 8-16 for illustrations depicting the use of the product output data type in the FIR Interpolation blocks of this block:

When you select `Inherit via internal rule`, the product output word length and fraction length are automatically set according to the following equations:

$$ideal\ product\ output\ word\ length\ = \\ input\ word\ length + FIR\ coefficients\ word\ length$$

$$ideal\ product\ output\ fraction\ length\ = \\ input\ fraction\ length + FIR\ coefficients\ fraction\ length$$

Note The actual product output word length may be equal to or greater than the calculated ideal product output word length, depending on the settings on the **Hardware Implementation** pane of the Configuration Parameters dialog box.

When you select `Same as input`, these characteristics will match those of the input to the block.

When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the product output, in bits.

When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the product output. This block requires power-of-two slope and a bias of zero.

Accumulator

As depicted above, inputs to the accumulator are cast to the accumulator data type. The output of the adder remains in the accumulator data type as each element of the input is added to it. Use this parameter to specify how you would like to designate this accumulator word and fraction lengths.

You also use this parameter to specify the accumulator word and fraction lengths resulting from a complex-complex multiplication in the FIR Interpolation blocks in this block. Refer to "Multiplication Data Types" on page 8-16 for more information:

When you select `Inherit via internal rule`, the accumulator word length and fraction length are automatically set according to the following equations:

$$
\begin{aligned}
ideal\ accumulator\ word\ length\ &=\ product\ output\ word\ length \\
&+ \mathrm{floor}(\log 2(number\ of\ accumulations)) + 1
\end{aligned}
$$

$$
ideal\ accumulator\ fraction\ length\ =\ product\ output\ fraction\ length
$$

where the number of accumulations is given by

$$
((number\ of\ coefficients/(interpolation\ factor)) - 1)
$$
if either the coefficients or inputs are real
$$
number\ of\ coefficients/(interpolation\ factor)
$$
if both the coefficients and inputs are complex

Note The actual accumulator word length may be equal to or greater than the calculated ideal product output word length, depending on the settings on the **Hardware Implementation** pane of the Configuration Parameters dialog box.

When you select `Same as product output`, these characteristics will match those of the product output

When you select `Same as input`, these characteristics will match those of the input to the block.

When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the accumulator, in bits.

When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the accumulator. This block requires power-of-two slope and a bias of zero.

Output

Choose how you will specify the output word length and fraction length of the FIR Interpolation blocks, as well as of the final overall filter output:

When you select `Same as accumulator`, these characteristics will match those of the accumulator.

A special case occurs when `Inherit via internal rule` is specified for **Accumulator**, and block inputs and coefficients are complex. In that case, the output word length will be one less than the accumulator word length.

When you select `Same as product output`, these characteristics will match those of the product output

When you select `Same as input`, these characteristics will match those of the input to the block.

When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the output, in bits.

When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the output. This block requires power-of-two slope and a bias of zero.

Two-Channel Synthesis Subband Filter

References   Fliege, N. J. *Multirate Digital Signal Processing: Multirate Systems, Filter Banks, Wavelets*. West Sussex, England: John Wiley & Sons, 1994.

Strang, G. and T. Nguyen. *Wavelets and Filter Banks*. Wellesley, MA: Wellesley-Cambridge Press, 1996.

Vaidyanathan, P. P. *Multirate Systems and Filter Banks*. Englewood Cliffs, NJ: Prentice Hall, 1993.

Supported Data Types
- Double-precision floating point
- Single-precision floating point
- Fixed point (signed only)
- 8-, 16-, and 32-bit signed integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Dyadic Synthesis Filter Bank	Signal Processing Blockset
FIR Interpolation	Signal Processing Blockset
Two-Channel Analysis Subband Filter	Signal Processing Blockset
fir1	Signal Processing Toolbox
fir2	Signal Processing Toolbox
firls	Signal Processing Toolbox
wfilters	Wavelet Toolbox

For related information, see "Multirate Filters" on page 3-66.

Purpose   Unbuffer input frame into sequence of scalar outputs

Library   Signal Management / Buffers

Description   The Unbuffer block unbuffers an M_i-by-N frame-based input into a
1-by-N sample-based output. That is, inputs are unbuffered *row-wise*
so that each matrix row becomes an independent time-sample in the
output. The rate at which the block receives inputs is generally less
than the rate at which the block produces outputs.
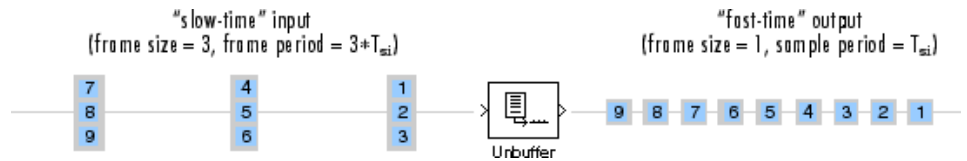
The block adjusts the output rate so that the *sample period* is the same
at both the input and output, $T_{so}=T_{si}$. Therefore, the output sample
period for an input of frame size M_i and frame period T_{fi} is T_{fi}/M_i,
which represents a *rate* M_i times higher than the input frame rate.
In the example above, the block receives inputs only once every three
sample periods, but produces an output once every sample period. To
rebuffer frame-based inputs to a larger or smaller frame size, use the
Buffer block.

In the model below, the block unbuffers a four-channel frame-based
input with frame size 3. The **Initial conditions** parameter is set
to zero and the tasking mode is set to multitasking, so the first three
outputs are zero vectors.

Unbuffer
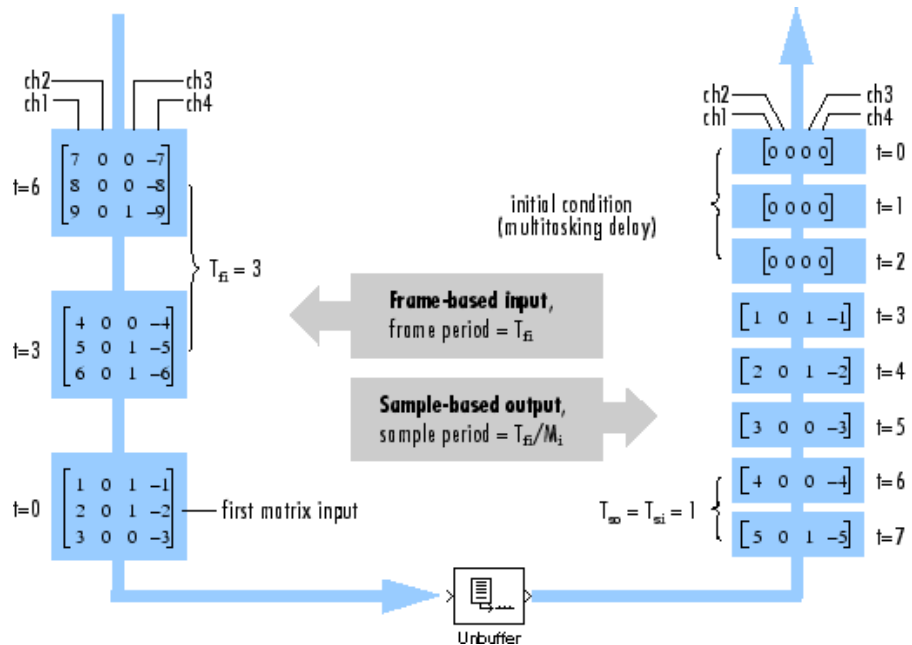
Zero Latency

The Unbuffer block has *zero tasking latency* in the Simulink single-tasking mode. Zero tasking latency means that the first input sample (received at *t*=0) appears as the first output sample.

Nonzero Latency

For *multitasking* operation, the Unbuffer block's buffer is initialized with the value specified by the **Initial condition** parameter, and the block begins unbuffering this frame at the start of the simulation. Inputs to the block are therefore delayed by one buffer length, or M_i samples.

The **Initial condition** parameter can be one of the following:

- A scalar to be repeated for the first M_i output samples of every channel

- A length-M_i vector containing the values of the first M_i output samples for every channel

- An M_i-by-N matrix containing the values of the first M_i output samples in each of N channels

---

Note For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" on page 2-56 and "Models with Multiple Sample Rates" in the Real-Time Workshop User's Guide documentation.

Dialog Box
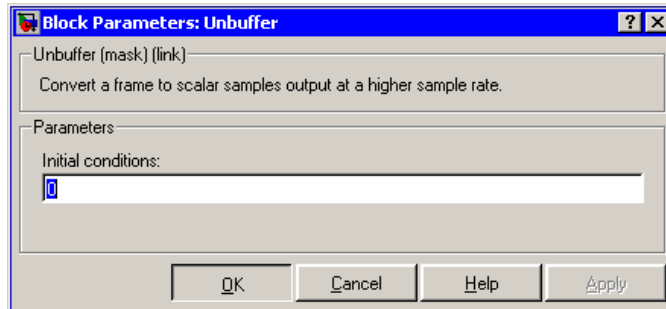
Initial conditions

The value of the block's initial output for cases of nonzero latency; a scalar, vector, or matrix.

Unbuffer

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Custom data types • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Custom data types • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Buffer   Signal Processing Blockset

See "Unbuffering Frame-Based Signals into Sample-Based Signals" on page 2-43 for related information.

Purpose   Decode integer input into floating-point output

Library   Quantizers
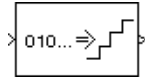
Description   The Uniform Decoder block performs the inverse operation of the
Uniform Encoder block, and reconstructs quantized floating-point
values from encoded integer input. The block adheres to the definition
for uniform decoding specified in ITU-T Recommendation G.701.

Inputs can be real or complex values of the following six integer data
types: uint8, uint16, uint32, int8, int16, or int32.

The block first casts the integer input values to floating-point values,
and then uniquely maps (decodes) them to one of 2^B uniformly spaced
floating-point values in the range $[-V, (1-2^{1-B})V]$, where you specify
B in the **Bits** parameter (as an integer between 2 and 32) and V is a
floating-point value specified by the **Peak** parameter. The smallest
input value representable by B bits (0 for an unsigned input data type;
-2^{B-1} for a signed input data type) is mapped to the value -V. The largest
input value representable by B bits (2^B-1 for an unsigned input data
type; 2^{B-1}-1 for a signed input data type) is mapped to the value $(1-2^{1-B})V$.
Intermediate input values are linearly mapped to the intermediate
values in the range $[-V, (1-2^{1-B})V]$.

To correctly decode values encoded by the Uniform Encoder block, the
Bits and **Peak** parameters of the Uniform Decoder block should be set
to the same values as the **Bits** and **Peak** parameters of the Uniform
Encoder block. The **Overflow mode** parameter specifies the Uniform
Decoder block's behavior when the integer input is outside the range
representable by B bits. When you select **Saturate**, *unsigned* input
values greater than 2^B-1 saturate at 2^B-1; *signed* input values greater
than 2^{B-1}-1 or less than -2^{B-1} saturate at those limits. The real and
imaginary components of complex inputs saturate independently.

When you select **Wrap**, *unsigned* input values, u, greater than 2^B-1 are
wrapped back into the range $[0, 2^B$-1$]$ using mod-2^B arithmetic.

```
u = mod(u,2^B)        % Equivalent MATLAB code
```

Uniform Decoder

Signed input values, u, greater than $2^{B-1}-1$ or less than -2^{B-1} are wrapped back into that range using mod-2^B arithmetic.

```
u = (mod(u+2^B/2,2^B)-(2^B/2))        % Equivalent MATLAB code
```

The real and imaginary components of complex inputs wrap independently.

The **Output type** parameter specifies whether the decoded floating-point output is single or double precision. Either level of output precision can be used with any of the six integer input data types.

Examples Consider a Uniform Decoder block with the following parameter settings:

- **Peak** = 2
- **Bits** = 3

The input to the block is the uint8 output of a Uniform Encoder block with comparable settings: **Peak** = 2, **Bits** = 3, and **Output type** = Unsigned. (Comparable settings ensure that inputs to the Uniform Decoder block do not saturate or wrap. See the example on the Uniform Encoder block reference page for more about these settings.)

The real and complex components of each input are independently mapped to one of 2^3 distinct levels in the range [-2.0,1.5].

```
0  is mapped to  -2.0
1  is mapped to  -1.5
2  is mapped to  -1.0
3  is mapped to  -0.5
4  is mapped to   0.0
5  is mapped to   0.5
6  is mapped to   1.0
7  is mapped to   1.5
```

Dialog Box
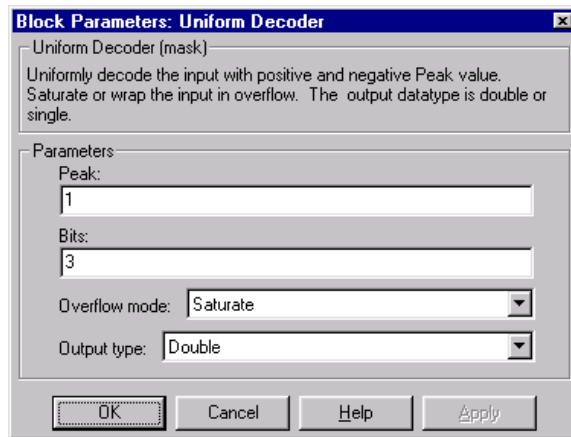
Peak

    The largest amplitude represented in the encoded input. To correctly decode values encoded with the Uniform Encoder block, set the **Peak** parameters in both blocks to the same value.

Bits

    The number of input bits, B, used to encode the data. (This can be less than the total number of bits supplied by the input data type.) To correctly decode values encoded with the Uniform Encoder block, set the **Bits** parameters in both blocks to the same value.

Overflow mode

    The block's behavior when the integer input is outside the range representable by B bits. Out-of-range inputs can either saturate at the extreme value, or wrap back into range.

Output type

    The precision of the floating-point output, single or double.

References   *General Aspects of Digital Transmission Systems: Vocabulary of Digital Transmission and Multiplexing, and Pulse Code Modulation (PCM) Terms,* International Telecommunication Union, ITU-T Recommendation G.701, March, 1993

Uniform Decoder

Supported Data Types

- Double-precision floating point

- Single-precision floating point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

|	
Data Type Conversion	Simulink
Quantizer	Simulink
Scalar Quantizer Decoder	Signal Processing Blockset
Uniform Encoder	Signal Processing Blockset
udecode	Signal Processing Toolbox
uencode	Signal Processing Toolbox

Purpose   Quantize and encode floating-point input into integer output

Library   Quantizers

Description

The Uniform Encoder block performs the following two operations on each floating-point sample in the input vector or matrix:
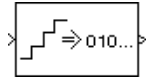
1 Quantizes the value using the same precision

2 Encodes the quantized floating-point value to an integer value

In the first step, the block quantizes an input value to one of 2^B uniformly spaced levels in the range [-V, $(1-2^{1-B})$V], where you specify B in the **Bits** parameter and you specify V in the **Peak** parameter. The quantization process rounds both positive and negative inputs *downward* to the nearest quantization level, with the exception of those that fall exactly on a quantization boundary. The real and imaginary components of complex inputs are quantized independently.

The number of bits, B, can be any integer value between 2 and 32, inclusive. Inputs greater than $(1-2^{1-B})$V or less than -V saturate at those respective values. The real and imaginary components of complex inputs saturate independently.

In the second step, the quantized floating-point value is uniquely mapped (encoded) to one of 2^B integer values. When the **Output type** is set to Unsigned integer, the smallest quantized floating-point value, -V, is mapped to the integer 0, and the largest quantized floating-point value, $(1-2^{1-B})$V, is mapped to the integer 2^B-1. Intermediate quantized floating-point values are linearly (uniformly) mapped to the intermediate integers in the range $[0, 2^B-1]$. For efficiency, the block automatically selects an *unsigned* output data type (uint8, uint16, or uint32) with the minimum number of bits equal to or greater than B.

When the **Output type** is set to Signed integer, the smallest quantized floating-point value, -V, is mapped to the integer -2^{B-1}, and the largest quantized floating-point value, $(1-2^{1-B})$V, is mapped to the

Uniform Encoder

integer 2^{B-1}-1. Intermediate quantized floating-point values are linearly mapped to the intermediate integers in the range [-2^{B-1}, 2^{B-1}-1]. The block automatically selects a *signed* output data type (int8, int16, or int32) with the minimum number of bits equal to or greater than B.

Inputs can be real or complex, double or single precision. The output data types that the block uses are shown in the table below. Note that most of the blocks in the Signal Processing Blockset accept only double-precision inputs. Use the Simulink Data Type Conversion block to convert integer data types to double precision. See "Working with Data Types" in the Simulink documentation for a complete discussion of data types, as well as a list of Simulink blocks capable of reduced-precision operations.

Bits	Unsigned Integer	Signed Integer
2 to 8	uint8	int8
9 to 16	uint16	int16
17 to 32	uint32	int32

The Uniform Encoder block operations adhere to the definition for uniform encoding specified in ITU-T Recommendation G.701.

Examples   The following figure illustrates uniform encoding with the following parameter settings:

- **Peak** = 2
- **Bits** = 3
- **Output type** = Unsigned
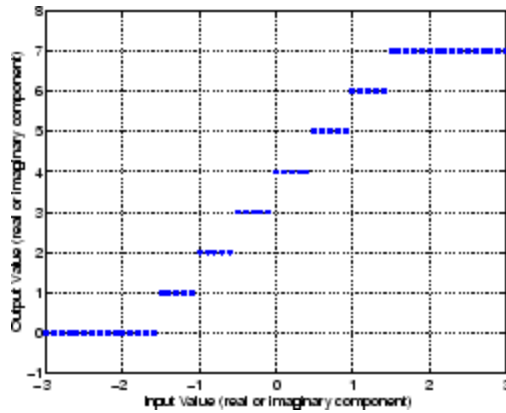
The real and complex components of each input (horizontal axis) are independently quantized to one of 2^3 distinct levels in the range [-2,1.5] and then mapped to one of 2^3 integer values in the range [0,7].

```
-2.0  is mapped to  0
-1.5  is mapped to  1
-1.0  is mapped to  2
-0.5  is mapped to  3
 0.0  is mapped to  4
 0.5  is mapped to  5
 1.0  is mapped to  6
 1.5  is mapped to  7
```

The table below shows the results for a few particular inputs.

Input	Quantized Input	Output	Notes
1.6	1.5+0.0i	7+4i	
-0.4	-0.5+0.0i	3+4i	

Uniform Encoder

Input	Quantized Input	Output	Notes
-3.2	-2.0+0.0i	4i	Saturation (real)
0.4-1.2i	0.0-1.5i	4+i	
0.4-6.0i	0.0-2.0i	4	Saturation (imaginary)
-4.2+3.5i	-2.0+2.0i	7i	Saturation (real and imaginary)

The output data type is automatically set to uint8, the most efficient format for this input range.

Dialog Box
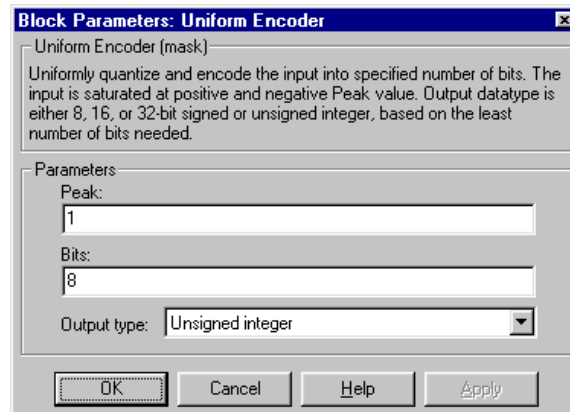
Peak

The largest input amplitude to be encoded, V. Real or imaginary input values greater than $(1-2^{1-B})$V or less than -V saturate (independently for complex inputs) at those limits.

Bits

> The number of levels at which to quantize the floating-point input. (Also the number of bits needed to represent the integer output.)

Output type

> The data type of the block's output, `Unsigned integer` or `Signed integer`. Unsigned outputs are `uint8`, `uint16`, or `uint32`, while signed outputs are `int8`, `int16`, or `int32`.

References   *General Aspects of Digital Transmission Systems: Vocabulary of Digital Transmission and Multiplexing, and Pulse Code Modulation (PCM) Terms,* International Telecommunication Union, ITU-T Recommendation G.701, March, 1993

Supported Data Types

- Double-precision floating point

- Single-precision floating point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Data Type Conversion	Simulink
Quantizer	Simulink
Scalar Quantizer Decoder	Signal Processing Blockset
Uniform Decoder	Signal Processing Blockset
udecode	Signal Processing Toolbox
uencode	Signal Processing Toolbox

Unwrap

Purpose　　　Unwrap signal phase

Library　　　Signal Operations

Description　The Unwrap block unwraps each input channel by adding or subtracting appropriate multiples of 2π to each channel element. The input can be any matrix or 1-D vector, and must have radian phase entries. The block recognizes phase discontinuities larger than the **Tolerance** parameter setting.

The block preserves the input size, dimension, and frame status, and the output port rate equals the input port rate. For a detailed discussion of the Unwrap block, see other sections of this reference page.

Sections of This Reference Page

- "Acceptable Inputs and Corresponding Output Characteristics" on page 10-1112
- "The Two Unwrap Modes" on page 10-1114
- "Unwrap Method" on page 10-1117
- "Definition of Phase Unwrap" on page 10-1118

Acceptable Inputs and Corresponding Output Characteristics

The Unwrap block preserves the input size, dimension, and frame status, and the output port rate equals the input port rate.

Characteristics of Valid Input	Characteristics of Corresponding Output
Input elements must be phase values in radians.	Output elements are phase values in radians.
Sample- or frame-based	Same frame status as input
M-by-N 2-D matrix or a 1-D vector	Same size and dimension as input
	Output port rate = input port rate

Unwrap

The Two Unwrap Modes

You must specify the unwrap mode by setting the parameter, **Do not unwrap phase discontinuities between successive frames**. The unwrap modes are summarized in the next table.

Two Unwrap Modes	
In both unwrap modes, the block adds $2\pi k$ to each input channel's elements, where it updates *k* at each phase discontinuity. (For more on the updating of *k*, see "Unwrap Method" on page 10-1117.) The number of times that *k* is reset to 0 depends on the unwrap mode.	
Default Unwrap Mode: Initialize k to 0 for Only the First Input Frame	**Nondefault Unwrap Mode: Set k to 0 for Each Successive Input Matrix or Input Vector**
☐ Do not unwrap phase discontinuities between successive frames	☑ Do not unwrap phase discontinuities between successive frames
In this mode, *k* is initialized to 0 for only the first input matrix or input vector. As *k* gets updated, the value of *k* is retained between successive input matrices or input vectors. That is, the block unwraps each input's channel by considering phase discontinuities in all previous frames and the current frame.	In this mode, *k* is reset to 0 for each successive input matrix or input vector. As *k* gets updated, the value of *k* is only retained within the current input matrix or vector. That is, the block unwraps each input's channel by considering phase discontinuities in the current input matrix or input vector only, ignoring discontinuities in previous inputs.

Two Unwrap Modes	
In both unwrap modes, the block adds $2\pi k$ to each input channel's elements, where it updates *k* at each phase discontinuity. (For more on the updating of *k*, see "Unwrap Method" on page 10-1117.) The number of times that *k* is reset to 0 depends on the unwrap mode.	
Default Unwrap Mode: Initialize k to 0 for Only the First Input Frame	**Nondefault Unwrap Mode: Set k to 0 for Each Successive Input Matrix or Input Vector**
In this mode, the block unwraps the columns or each individual element of the input: • Frame-based inputs – unwrap columns • Sample-based inputs – unwrap each element of the input. • 1-D vector inputs – treat as frame-based column	In this mode, the block unwraps the columns or rows of the input: • Frame-based inputs – unwrap columns • Sample-based nonrow inputs – unwrap columns • Sample-based row vector inputs – unwrap the row. • 1-D vector inputs – treat as frame-based column
See the following diagrams.	See the following diagrams.

The following diagrams illustrate how the two unwrap modes operate on various inputs.

Unwrap

Default Unwrap Mode Operation:

☐ Do not unwrap phase discontinuities between successive frames

Frame-Based Inputs

The block treats each input column as an independent channel. It unwraps by treating Channel 1 of Frame 2 as a continuation of Channel 1 of Frame 1.
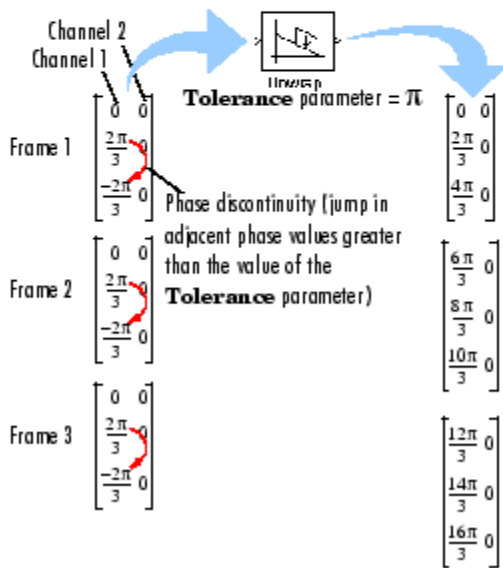
Sample-Based Inputs

The block treats each element of the input matrix as an independent channel. (The first sample in Channel 1 is in the upper left corner of the Sample 1 matrix. The second sample of Channel 1 is in the corresponding corner of the Sample 2 matrix, and so on.)
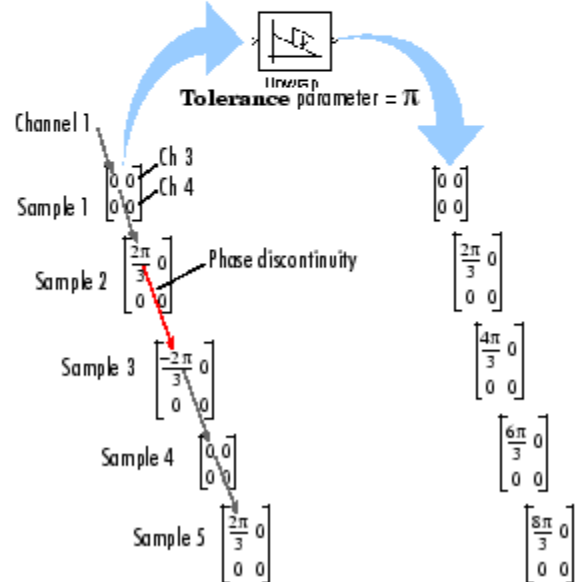
Nondefault Unwrap Mode Operation: ☑ Do not unwrap phase discontinuities between successive frames

Frame-Based Inputs and Sample-Based (Nonrow) Inputs

The block unwraps each column, treating each input matrix as completely unrelated to the other input matrices.
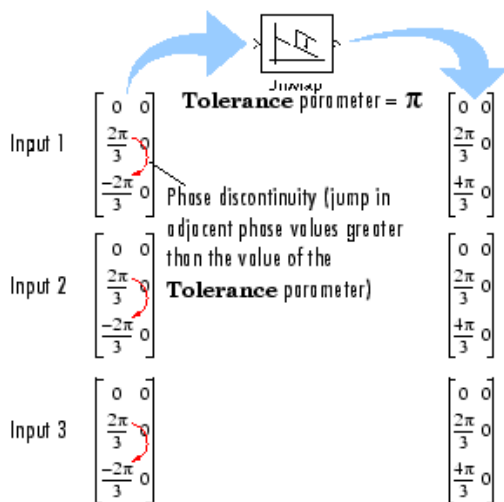
Sample-Based Row Vector Inputs

The block unwraps each row, treating each input row vector as completely independent of the other input row vectors.
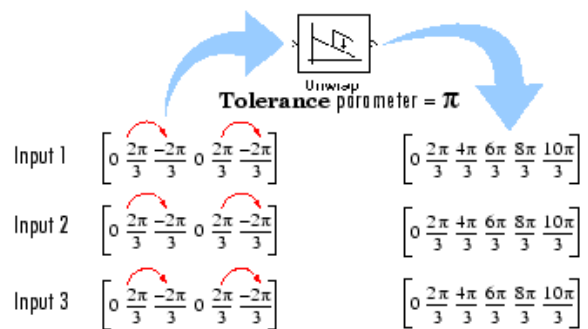
Unwrap Method

The Unwrap block unwraps each channel of its input matrix or input vector by adding $2\pi k$ to each successive channel element, and updating k at each *phase jump*. See the following steps to the unwrap method for details.

Unwrap

Definition of Phase Unwrap

Algorithms that compute the phase of a signal often only output phases between $-\pi$ and π. For instance, such algorithms compute the phase of $\sin(2\pi + 3)$ to be 3, since $\sin(3) = \sin(2\pi + 3)$, and since the actual

phase, $2\pi + 3$, is not between $-\pi$ and π. Such algorithms compute the phases of $\sin(-4\pi + 3)$ and $\sin(16\pi + 3)$ to be 3 as well.

Phase unwrap or unwrap is a process often used to reconstruct a signal's original phase. Unwrap algorithms add appropriate multiples of 2π to each phase input to restore original phase values, as illustrated in the following diagram. For more on phase unwrap, see the previous section, "Unwrap Method" on page 10-1117.
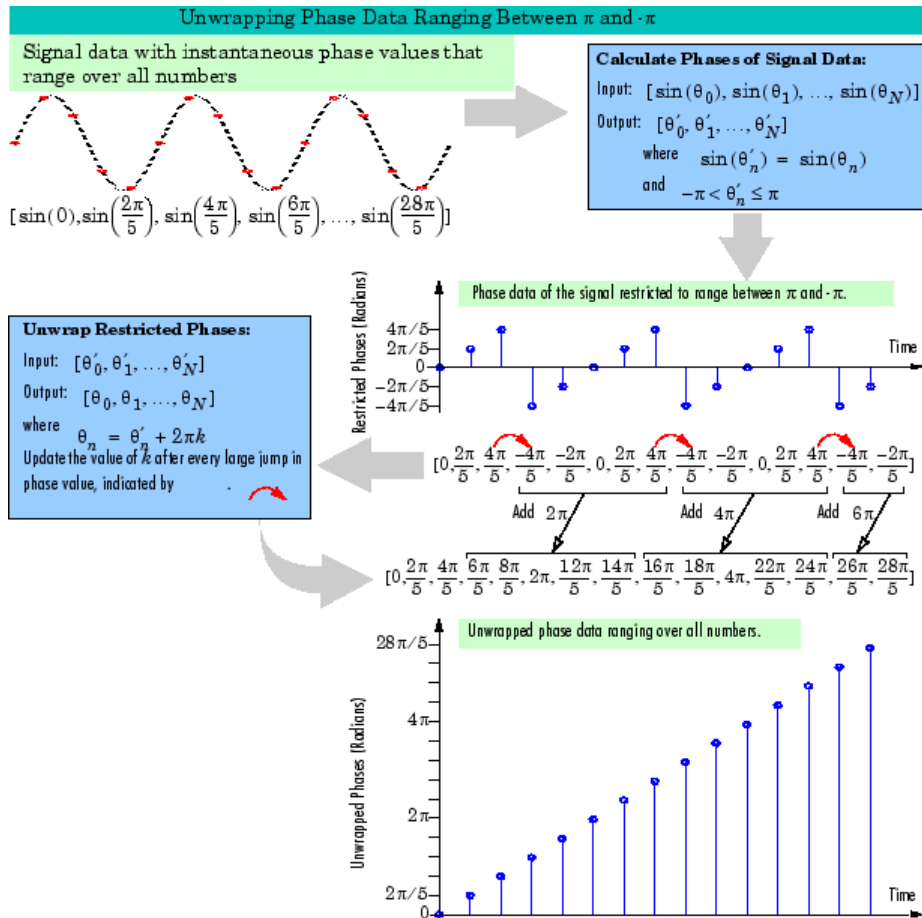
Unwrap
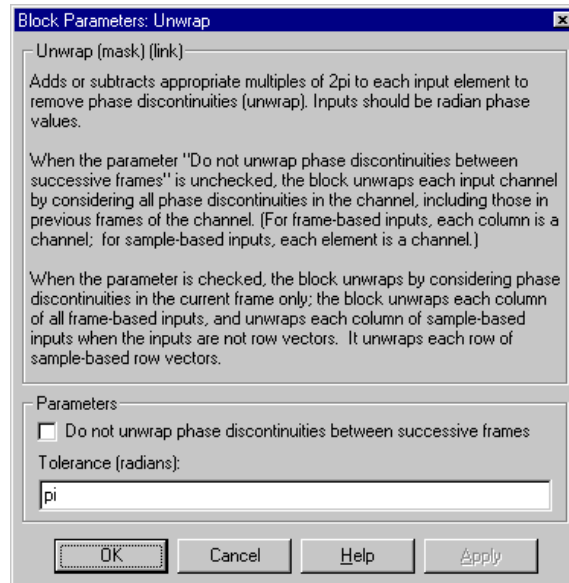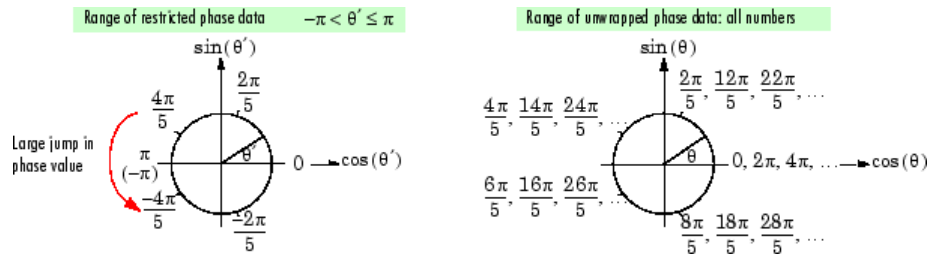
Do not unwrap phase discontinuities between successive frames

When this parameter is cleared, the block unwraps each input's channels (the input channels are the columns of frame-based inputs and each element of sample-based inputs). When you select this parameter, the block unwraps each row of sample-based row vector inputs, and unwraps the columns of all other inputs, where each input matrix or input vector is treated as completely unrelated to the other input matrices or input vectors. 1-D vector

inputs are always treated as frame-based column vectors. See "The Two Unwrap Modes" on page 10-1114.

Tolerance

The jump size that the block recognizes as a true phase discontinuity. The default is set to π (rather than a smaller value) to avoid altering legitimate signal features. To increase the block's sensitivity, set **Tolerance** to a value slightly less than π.

Supported Data Types

- Double-precision floating point
- Single-precision floating point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

unwrap   MATLAB

Upsample

Purpose   Resample input at higher rate by inserting zeros

Library   Signal Operations

Description   The Upsample block resamples each channel of the M_i-by-N input at
a rate L times higher than the input sample rate by inserting L-1
zeros between consecutive samples. You specify the integer L in the
Upsample factor parameter. The **Sample offset** parameter delays
the output samples by an integer number of sample periods D, where
$0 \leq D < (L-1)$, so that any of the L possible output phases can be selected.

This block supports triggered subsystems if, for **Frame-based mode**,
you select Maintain input frame rate.

Sample-Based Operation

When the input is sample based, the block treats each of the M*N
matrix elements as an independent channel, and upsamples each
channel over time. The **Frame-based mode** parameter must be set
to Maintain input frame size. The output sample rate is L times
higher than the input sample rate ($T_{so} = T_{si}/L$), and the input and output
sizes are identical.

Frame-Based Operation

When the input is frame based, the block treats each of the N input
columns as a frame containing M_i sequential time samples from an
independent channel. The block upsamples each channel independently
by inserting L-1 rows of zeros between each row in the input matrix.
The **Frame-based mode** parameter determines how the block adjusts
the rate at the output to accommodate the added rows. There are two
available options:

- Maintain input frame size

  The block generates the output at the faster (upsampled) rate by
  using a proportionally shorter frame *period* at the output port than
  at the input port. For upsampling by a factor of L, the output frame
  period is L times shorter than the input frame period ($T_{fo} = T_{fi}/L$), but
  the input and output frame sizes are equal.

The model below shows a single-channel input with a frame period of 1 second being upsampled by a factor of 4 to a frame period of 0.25 second. The input and output frame sizes are identical.
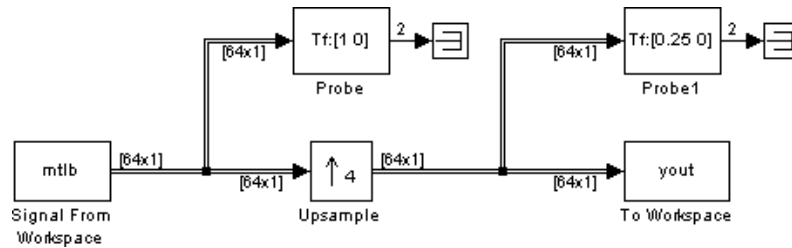
- Maintain input frame rate

  The block generates the output at the faster (upsampled) rate by using a proportionally larger frame *size* than the input. For upsampling by a factor of L, the output frame size is L times larger than the input frame size ($M_o = M_i*L$), but the input and output frame rates are equal.

  The model below shows a single-channel input of frame size 16 being upsampled by a factor of 4 to a frame size of 64. The input and output frame rates are identical.
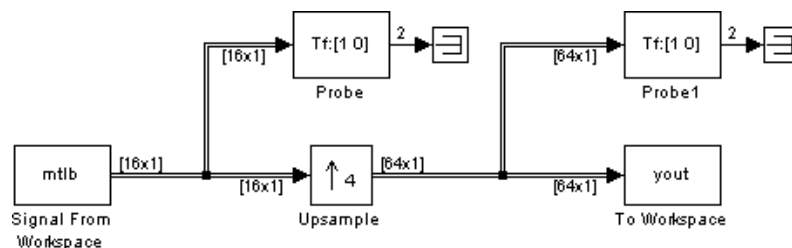
Zero Latency

The Upsample block has *zero tasking latency* for all single-rate operations. The block is single-rate for the particular combinations of sampling mode and parameter settings shown in the table below.

Upsample

Sampling Mode	Parameter Settings
Sample based	**Upsample factor** parameter, L, is 1.
Frame based	**Upsample factor** parameter, L, is 1, *or* **Frame-based mode** parameter is Maintain input frame rate.

The block also has zero latency for all multirate operations in the Simulink single-tasking mode.

Zero tasking latency means that the block propagates the first input (received at *t*=0) immediately following the D consecutive zeros specified by the **Sample offset** parameter. This output (D+1) is followed in turn by the L-1 inserted zeros and the next input sample. The **Initial condition** parameter value is not used.

Nonzero Latency

The Upsample block has tasking latency only for multirate operation in the Simulink multitasking mode:

- In sample-based mode, the initial condition for each channel appears as output sample D+1, and is followed by L-1 inserted zeros. The channel's first input appears as output sample D+L+1. The **Initial condition** value can be an M_i-by-N matrix containing one value for each channel, or a scalar to be applied to all signal channels.

- In frame-based mode, the first row of the initial condition matrix appears as output sample D+1, and is followed by L-1 inserted rows of zeros, the second row of the initial condition matrix, and so on. The first row of the first input matrix appears in the output as sample M_iL+D+1. The **Initial condition** value can be an M_i-by-N matrix, or a scalar to be repeated across all elements of the M_i-by-N matrix. See the example below for an illustration of this case.

Note For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" on page 2-56 and "Models with Multiple Sample Rates" in the Real-Time Workshop User's Guide documentation.

Examples

Construct the frame-based model shown below.

Adjust the block parameters as follows:

- Configure the Signal From Workspace block to generate a two-channel signal with frame size of 4 and sample period of 0.25. This represents an output frame period of 1 (0.25*4). The first channel should contain the positive ramp signal 1, 2, ..., 100, and the second channel should contain the negative ramp signal -1, -2, ..., -100.

  - **Signal** = [(1:100)' (-1:-1:-100)']
  - **Sample time** = 0.25
  - **Samples per frame** = 4

- Configure the Upsample block to upsample the two-channel input by increasing the output frame rate by a factor of 2 relative to the input frame rate. Set a sample offset of 1, and an initial condition matrix of

$$\begin{bmatrix} 11 & -11 \\ 12 & -12 \\ 13 & -13 \\ 14 & -14 \end{bmatrix}$$

- **Upsample factor** = 2
- **Sample offset** = 1
- **Initial condition** = [11 -11;12 -12;13 -13;14 -14]
- **Frame-based mode** = Maintain input frame size

- Configure the Probe blocks by clearing the **Probe width** and **Probe complex signal** check boxes (if desired).

This model is multirate because there are at least two distinct frame rates, as shown by the two Probe blocks. To run this model in the Simulink multitasking mode, open the Configuration Parameters dialog box. In the **Select** pane, click **Solver**. From the **Type** list, select Fixed-step, and from the **Solver** list, select discrete (no continuous states). From the **Tasking mode for periodic sample times** list, select MultiTasking. Also set the **Stop time** to 30.

Run the model and look at the output, yout. The first few samples of each channel are shown below.

```
yout =

     0      0
    11    -11
     0      0
    12    -12
     0      0
    13    -13
     0      0
    14    -14
     0      0
     1     -1
     0      0
```

```
2    -2
0     0
3    -3
0     0
4    -4
0     0
5    -5
0     0
```

Since we ran this frame-based multirate model in multitasking mode, the first row of the initial condition matrix appears as output sample 2 (that is, sample D+1, where D is the **Sample offset** value). It is followed by the other three initial condition rows, each separated by L-1 inserted rows of zeros, where L is the **Upsample factor** value of 2. The first row of the first input matrix appears in the output as sample 10 (that is, sample M_iL+D+1, where M_i is the input frame size).

Dialog Box
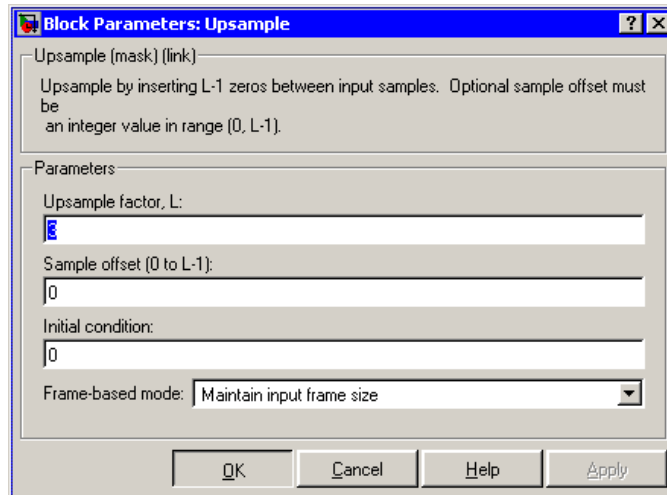
Upsample factor

    The integer factor, L, by which to increase the input sample rate.

Upsample

Sample offset

The sample offset, D, which must be an integer in the range [0,L-1].

Initial condition

The value with which the block is initialized for cases of nonzero latency, a scalar or matrix. This value (first row in frame-based mode) appears in the output as sample D+1.

Frame-based mode

For frame-based operation, the method by which to implement the upsampling: Maintain input frame size (that is, increase the frame rate), or Maintain input frame rate (that is, increase the frame size). The **Framing** parameter must be set to Maintain input frame size for sample-base inputs.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Custom data types • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Custom data types • Boolean • 8-, 16-, and 32-bit signed integers

Port	Supported Data Types
	• 8-, 16-, and 32-bit unsigned integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Downsample	Signal Processing Blockset
FIR Interpolation	Signal Processing Blockset
FIR Rate Conversion	Signal Processing Blockset
Repeat	Signal Processing Blockset

Variable Fractional Delay

Purpose   Delay input by time-varying fractional number of sample periods

Library   Signal Operations
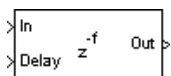
Description   The Variable Fractional Delay block delays each channel of the M_i-by-N input matrix, u, by a variable (possibly noninteger) number of sample intervals.

The block computes the value for each channel of the output based on the stored samples in memory most closely indexed by the Delay input, v, and the interpolation method specified by the *Mode* parameter. In Linear Interpolation mode, the block stores the D+1 most recent samples received at the In port for each channel, where D is the **Maximum delay**. In FIR Interpolation mode, the block stores the $D+P+1$ most recent samples received at the In port for each channel, where P is the **Interpolation filter half-length**.

See the Variable Integer Delay block for further discussion of how input samples are stored in the block's memory. The Variable Fractional Delay block differs only in the way that these stored sample are *accessed*; a fractional delay requires the computation of a value by interpolation from the nearby samples in memory.

Sample-Based Operation

When the input is sample based, the block treats each of the M_i*N matrix elements as an independent channel. The input to the Delay port, v, can be an M_i-by-N matrix of floating-point values in the range $0 \leq v \leq D$ that specifies the number of sample intervals to delay each channel of the input. It can also be a scalar floating-point value, $0 \leq v \leq D$, by which to equally delay all channels.

A 1-D vector input is treated as an M_i-by-1 matrix, and the output is 1-D.

The **Initial conditions** parameter specifies the values in the block's memory at the start of the simulation in the same manner as the Variable Integer Delay block. See the Variable Integer Delay block reference page for more information.

Frame-Based Operation

When the input is frame based, the block treats each of the *N* input columns as a frame containing M_i sequential time samples from an independent channel.

The input to the Delay port, *v*, contains floating-point values in the range $0 \le v \le D$ specifying the number of sample intervals to delay the current input. The input to the Delay port can be

- A scalar value by which to equally delay all channels

- An M_i-by-*N* matrix containing the number of sample intervals to delay *each* sample in *each* channel of the current input

- An M_i-by-1 matrix containing the number of sample intervals to delay each sample in *every* channel of the current input

- A 1-by-*N* matrix containing the number of sample intervals to delay *every* sample in each channel of the current input

For example, if *v* is the M_i-by-1 matrix `[v(1) v(2) ... v(Mi)]'`, the earliest sample in the current frame is delayed by `v(1)` fractional sample intervals, the following sample in the frame is delayed by `v(2)` fractional sample intervals, and so on. The set of fractional delays contained in *v* is applied identically to every channel of a multichannel input.

The **Initial conditions** parameter specifies the values in the block's memory at the start of the simulation in the same manner as the Variable Integer Delay block. See the Variable Integer Delay block reference page for more information.

Interpolation Modes

The delay value specified at the Delay port is used as an index into the block's memory, U, which stores the D+1 most recent samples received at the In port for each channel. For example, an integer delay of 5 on a scalar input sequence retrieves and outputs the fifth most recent input sample from the block's memory, U(6). Fractional delays are

computed by interpolating between stored samples; the two available interpolation modes are described below.

Linear Interpolation Mode

For noninteger delays, at each sample time the `Linear Interpolation` mode uses the two samples in memory nearest to the specified delay to compute a value for the sample at that time. If `v` is the specified fractional delay for a scalar input, the output sample, `y`, is computed as follows.

```
vi = floor(v)     % vi = integer delay
vf = v-vi         % vf = fractional delay
y = (1-vf)*U(vi) + vf*U(vi+1)
```

Delay values less than 0 are clipped to 0, and delay values greater than *D* are clipped to *D*, where *D* is the **Maximum delay**. Note that a delay value of 0 causes the block to pass through the current input sample, `U(1)`, in the same simulation step that it is received.

FIR Interpolation Mode

In `FIR Interpolation` mode, the block computes a value for the sample at the desired delay by applying an FIR filter of order $2P$ to the stored samples on either side of the desired delay, where *P* is the **Interpolation filter half-length**. For periodic signals, a larger value of *P* (that is, a higher order filter) yields a better estimate of the sample at the specified delay. A value between 4 and 6 for this parameter (that is, a 7th to 11th order filter) is usually adequate.

A vector of $2P$ filter tap weights is precomputed at the start of the simulation for each of *Q*-1 discrete points between input samples, where you specify *Q* in the **Interpolation points per input sample** parameter. For a delay corresponding to one of the *Q* interpolation points, the unique filter computed for that interpolation point is applied to obtain a value for the sample at the specified delay. For delay times that fall between interpolation points, the value computed at the nearest interpolation point is used. Since *Q* controls the number of locations where a unique interpolation filter is designed, a larger value results in a better estimate of the sample at a given delay.

Note that increasing the **Interpolation filter half length** (*P*) increases the number of computations performed per input sample, as well as the amount of memory needed to store the filter coefficients. Increasing the **Interpolation points per input sample** (*Q*) increases the simulation's memory requirements but does not affect the computational load per sample.

The **Normalized input bandwidth** parameter allows you to take advantage of the bandlimited frequency content of the input. For example, if you know that the input signal does not have frequency content above $F_s/4$, you can specify a value of 0.5 for the **Normalized input bandwidth** to constrain the frequency content of the output to that range.

(Each of the *Q* interpolation filters can be considered to correspond to one output phase of an "upsample-by-*Q*" FIR filter. In this view, the **Normalized input bandwidth** value is used to improve the stopband in critical regions, and to relax the stopband requirements in frequency regions where there is no signal energy.)

For delay values less than P/2-1, the output is computed using linear interpolation. Delay values greater than *D* are clipped to *D*, where *D* is the **Maximum delay**.

The block uses the intfilt function in the Signal Processing Toolbox to compute the FIR filters.

Note When the Variable Fractional Delay block is used in a feedback loop, at least one block with nonzero delay (for example, a Delay block with **Delay** > 0) should be included in the loop as well. This prevents the occurrence of an algebraic loop when the delay of the Variable Fractional Delay block is driven to zero.

Examples   The dspafxf demo illustrates an audio flanger system built around the Variable Fractional Delay block.

Variable Fractional Delay

Dialog Box

Block Parameters: Variable Fractional Delay

Variable Fractional Delay (mask)

Delay discrete-time input by a time-varying fractional number of sample periods, as specified by the "delay" input. The FIR filter is designed using "intfilt" from the Signal Processing Toolbox.

Parameters

Mode: FIR Interpolation

Maximum delay (samples):

100

Interpolation filter half-length:

4

Interpolation points per input sample:

10

Normalized input bandwidth (0 to 1):

1

Initial conditions:

0

OK   Cancel   Help   Apply

Mode
> The method by which to interpolate between adjacent stored samples to obtain a value for the sample indexed by the input at the Delay port.

Maximum delay
> The maximum delay that the block can produce, D. Delay input values exceeding this maximum are clipped at the maximum.

Interpolation filter half-length
> Half the number of input samples to use in the FIR interpolation filter.

Interpolation points per input sample
> The number of points per input sample, Q, at which a unique FIR interpolation filter is computed.

Normalized input bandwidth
> The bandwidth to which the interpolated output samples should be constrained. A value of 1 specifies half the sample frequency.

Initial conditions
> The values with which the block's memory is initialized. See the Variable Integer Delay block for more information.

Supported Data Types

- Double-precision floating point
- Single-precision floating point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Delay	Signal Processing Blockset
Unit Delay	Simulink
Variable Integer Delay	Signal Processing Blockset

Variable Integer Delay

Purpose　　　Delay input by time-varying integer number of sample periods

Library　　　Signal Operations

Description
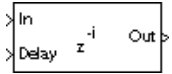
The Variable Integer Delay block delays the discrete-time input at the In port by the integer number of sample intervals specified by the input to the Delay port. The sample rate of the input signal at the Delay port must be the same as the sample rate of the input signal at the In port. When these sample rates are not the same, you need to insert a Zero-Order Hold or Rate Transition block in order to make the sample rates identical. The delay for a sample-based input sequence is a scalar value to uniformly delay every channel. The delay for a frame-based input sequence can be a scalar value to uniformly delay every sample in every channel, a vector containing one delay value for each sample in the input frame, or a vector containing one delay value for each channel in the input frame.

The delay values should be in the range of 0 to D, where D is the **Maximum delay**. Delay values greater than D or less than 0 are clipped to those respective values and noninteger delays are rounded to the nearest integer value.

The Variable Integer Delay block differs from the Delay block in the following ways.

Variable Integer Delay Block	Delay Block
The delay is provided as an input to the Delay port.	You specify the delay as a parameter setting in the dialog box.
Delay can vary with time; for example, for a frame-based input, the nth element's delay in the first input frame can differ from the nth element's delay in the second input frame.	Delay cannot vary with time; for example, for a frame-based input, the nth element's delay is the same for every input frame.

Sample-Based Operation

When the input is an *M*-by-*N* sample-based matrix, the block treats each of the *M**N* matrix elements as an independent channel, and applies the delay at the Delay port to each channel.

The Variable Integer Delay block stores the D+1 most recent samples received at the In port for each channel. At each sample time the block outputs the stored sample(s) indexed by the input to the Delay port.

For example, when the input to the In port, u, is a scalar signal, the block stores a vector, U, of the D+1 most recent signal samples. When the current input sample is U(1), the previous input sample is U(2), and so on, then the block's output is

```
y = U(v+1);    % Equivalent MATLAB code
```

where v is the input to the Delay port. Note that a delay value of 0 (v=0) causes the block to pass through the sample at the In port in the same simulation step that it is received. The block's memory is initialized to the **Initial conditions** value at the start of the simulation (see below).

The figure below shows the block output for a scalar ramp sequence at the In port, a **Maximum delay** of 5, an **Initial conditions** of 0, and a variety of different delays at the Delay port.

Variable Integer Delay
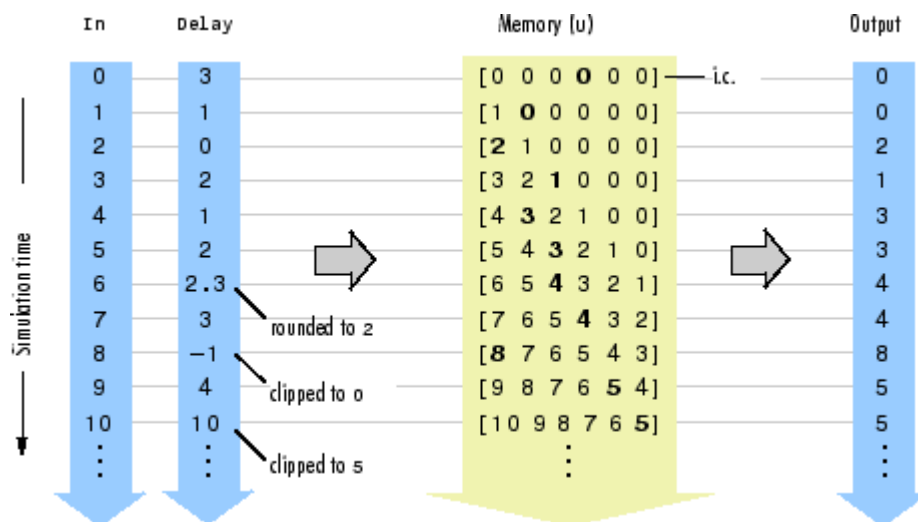
Note that the current input at each time-step is immediately stored in memory as U(1). This allows the current input to be available at the output for a delay of 0 (v=0).

The **Initial conditions** parameter specifies the values in the block's memory at the start of the simulation. Unlike the Delay block, the Variable Integer Delay block does not have a fixed *initial delay* period during which the initial conditions appear at the output. Instead, the initial conditions are propagated to the output only when they are indexed in memory by the value at the Delay port. Both fixed and time-varying initial conditions can be specified in a variety of ways to suit the dimensions of the input sequence.

Fixed Initial Conditions

The settings shown below specify *fixed* initial conditions. For a fixed initial condition, the block initializes each of D samples in memory to the value entered in the **Initial conditions** parameter. A fixed initial condition in sample-based mode can be specified in one of the following ways:

- *Scalar* value with which to initialize every sample of every channel in memory. For a general *M*-by-*N* input and the parameter settings below,

  Maximum delay (samples):
  100
  Initial conditions:
  0

  the block initializes 100 *M*-by-*N* matrices in memory with zeros.
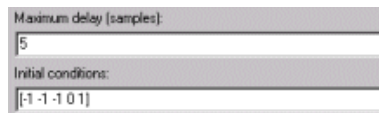
- *Array* of size *M*-by-*N*-by-*D*. In this case, you can specify different fixed initial conditions for each channel. See the *Array* bullet in "Time-Varying Initial Conditions" on page 10-1139 below for details.

Initial conditions cannot be specified by full matrices.

Time-Varying Initial Conditions

The following settings specify *time-varying* initial conditions. For a time-varying initial condition, the block initializes each of *D* samples in memory to one of the values entered in the **Initial conditions** parameter. This allows you to specify a unique output value for each sample in memory. A time-varying initial condition in sample-based mode can be specified in one of the following ways:

- *Vector* containing *D* elements with which to initialize memory samples U(2:D+1), where *D* is the **Maximum delay**. For a scalar input and the parameters shown below, the block initializes U(2:6) with values [-1, -1, -1, 0, 1].

  Maximum delay (samples):
  5
  Initial conditions:
  [-1 -1 -1 0 1]

- *Array* of dimension *M*-by-*N*-by-*D* with which to initialize memory samples U(2:D+1), where *D* is the **Maximum delay** and *M* and *N* are the number of rows and columns, respectively, in the input

Variable Integer Delay

matrix. For a 2-by-3 input and the parameters below, the block initializes memory locations U(2:5) with values

$$U(2) = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}, U(3) = \begin{bmatrix} 2 & 2 & 2 \\ 2 & 2 & 2 \end{bmatrix}, U(4) = \begin{bmatrix} 3 & 3 & 3 \\ 3 & 3 & 3 \end{bmatrix}, U(5) = \begin{bmatrix} 4 & 4 & 4 \\ 4 & 4 & 4 \end{bmatrix}$$

Maximum delay (samples):

4

Initial conditions:

cat(3, [1 1 1; 1 1 1], [2 2 2; 2 2 2], [3 3 3; 3 3 3], [4 4 4; 4 4 4])
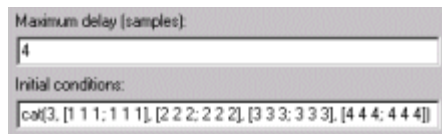
An array initial condition can only be used with matrix inputs.

Initial conditions cannot be specified by full matrices.

Frame-Based Operation

When the input is an *M*-by-*N* frame-based matrix, the block treats each of the *N* input columns as a frame containing *M* sequential time samples from an independent channel.

In frame-based mode, the input at the Delay port can be a scalar value to uniformly delay every sample in every channel. It can also be a length- vector, v = [v(1) v(2) ... v(M)], containing one delay for each sample in the input frame(s). The set of delays contained in vector v is applied identically to every channel of a multichannel input. The Delay port entry can also be a length- vector, containing one delay for each channel.

Vector v *does not* specify when the samples in the current input frame will appear in the output. Rather, v indicates which *previous* input samples (stored in memory) should be included in the current output frame. The first sample in the current output frame is the input sample v(1) intervals earlier in the sequence, the second sample in the current output frame is the input sample v(2) intervals earlier in the sequence, and so on.

The illustration below shows how this works for an input with a sample period of 1 and frame size of 4. The **Maximum delay** (Dmax) is 5, and

the **Initial conditions** parameter is set to -1. The delay input changes from [1 3 0 5] to [2 0 0 2] after the second input frame. Note that the samples in each output frame are the values in memory indexed by the elements of v.

```
y(1) = U(v(1)+1)
y(2) = U(v(2)+1)
y(3) = U(v(3)+1)
y(4) = U(v(4)+1)
```
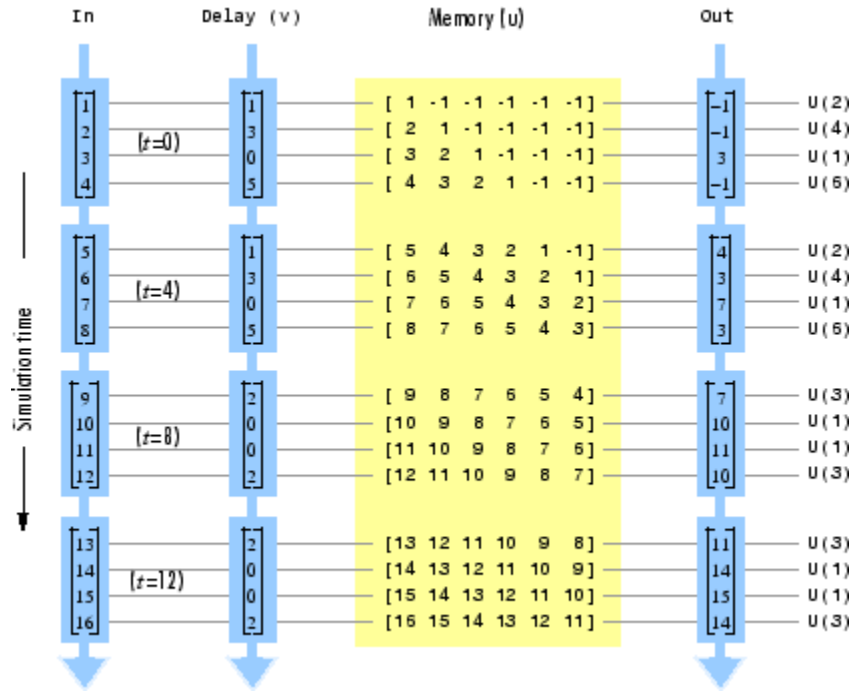

The **Initial conditions** parameter specifies the values in the block's memory at the start of the simulation. Both fixed and time-varying initial conditions can be specified.

Fixed Initial Conditions

The settings shown below specify *fixed* initial conditions. For a fixed initial condition, the block initializes each of D samples in memory to the value entered in the **Initial conditions** parameter. A fixed initial condition in frame-based mode can be one of the following:

- *Scalar* value with which to initialize every sample of every channel in memory. For a general M-by-N input with the parameter settings below, the block initializes five samples in memory with zeros.

  Maximum delay (samples):
  5

  Initial conditions:
  0

- *Array* of size 1-by-N-by-D. In this case, you can specify different fixed initial conditions for each channel. See the *Array* bullet in "Time-Varying Initial Conditions" on page 10-1142 below for details.

Initial conditions cannot be specified by full matrices.

Time-Varying Initial Conditions

The following setting specifies a *time-varying* initial condition. For a time-varying initial condition, the block initializes each of D samples in memory to one of the values entered in the **Initial conditions** parameter. This allows you to specify a unique output value for each sample in memory. A time-varying initial condition in frame-based mode can be specified in the following ways:

- *Vector* of dimensions 1-by-D. In this case, all channels have the same set of time-varying initial conditions specified by the entries of the vector. For the ramp input [100; 100]' with a frame size of 4, delay of 5, and the parameter settings below, the block outputs the following sequence of frames at the start of the simulation.

$$
\begin{bmatrix} -1 & -1 \\ -2 & -2 \\ -3 & -3 \\ -4 & -4 \end{bmatrix}, \begin{bmatrix} -5 & -5 \\ 1 & 1 \\ 2 & 2 \\ 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 \\ 5 & 5 \\ 6 & 6 \\ 7 & 7 \end{bmatrix}, \cdots
$$

Maximum delay (samples):

```
5
```

Initial conditions:

```
[-1 -2 -3 -4 -5]
```

- *Array* of size 1-by-*N*-by-*D*. In this case, you can specify different time-varying initial conditions for each channel. For the ramp input [100; 100]' with a frame size of 4, delay of 5, and the parameter settings below, the block outputs the following sequence of frames at the start of the simulation.

$$
\begin{bmatrix} -1 & -11 \\ -2 & -22 \\ -3 & -33 \\ -4 & -44 \end{bmatrix}, \begin{bmatrix} -5 & -55 \\ 1 & 1 \\ 2 & 2 \\ 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 \\ 5 & 5 \\ 6 & 6 \\ 7 & 7 \end{bmatrix}, \cdots
$$

Maximum delay (samples):

```
5
```

Initial conditions:

```
cat(3, [-1 -11], [-2 -22], [-3 -33], [-4 -44], [-5 -55])
```

Note that by specifying a 1-by-*N*-by-*D* initial condition array such that each 1-by-*N* vector entry is identical, you can implement different *fixed* initial conditions for each channel.

Initial conditions cannot be specified by full matrices.

Variable Integer Delay

Dialog Box
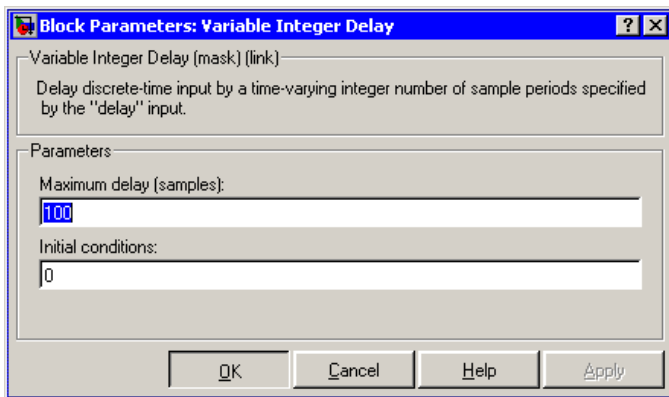
Maximum delay

> The maximum delay that the block can produce for any sample. Delay input values exceeding this maximum are clipped at the maximum.

Initial conditions

> The values with which the block's memory is initialized.

Supported Data Types

Port	Supported Data Types
In	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Custom data types
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

Port	Supported Data Types
Delay	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Custom data types • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Out	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Custom data types • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Delay	Signal Processing Blockset
Variable Fractional Delay	Signal Processing Blockset

Variable Selector

Purpose   Select subset of rows or columns from input

Library   Signal Management / Indexing
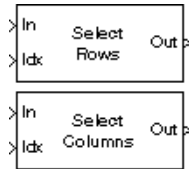
Description   The Variable Selector block extracts a subset of rows or columns from
the *M*-by-*N* input matrix u at each input port. You specify the number
of input and output ports in the **Number of input signals** parameter.

When the **Select** parameter is set to Rows, the Variable Selector block
extracts rows from each input matrix, while if the **Select** parameter
is set to Columns, the block extracts columns.

When the **Selector mode** parameter is set to Variable, the length-*L*
vector input to the Idx port selects *L* rows or columns of each input to
pass through to the output. The elements of the indexing vector can be
updated at each sample time, but the vector length must remain the
same throughout the simulation.

When the **Selector mode** parameter is set to Fixed, the Idx port is
disabled, and the length-*L* vector specified in the **Elements** parameter
selects *L* rows or columns of each input to pass through to the output.
The **Elements** parameter is tunable, so you can change the values
of the indexing vector elements at any time during the simulation;
however, the vector length must remain the same.

For both variable and fixed indexing modes, the row selection operation
is equivalent to

```
y = u(idx,:)      % Equivalent MATLAB code
```

and the column selection operation is equivalent to

```
y = u(:,idx)      % Equivalent MATLAB code
```

where idx is the length-*L* indexing vector. The row selection output size
is *L*-by-*N* and the column selection output size is *M*-by-*L*. Input rows or
columns can appear any number of times in the output, or not at all.

When the input is a 1-*D* vector, the **Select** parameter is ignored; the output is a 1-*D* vector of length *L* containing those elements specified by the length-*L* indexing vector.

When an element of the indexing vector references a nonexistent row or column of the input, the block reacts with the behavior specified by the **Invalid index** parameter. The following options are available:

- `Clip index` — Clip the index to the nearest valid value, and *do not* issue an alert. Example: For a 64-by-*N* input, an index of 72 is clipped to 64; an index of -2 is clipped to 1.

- `Clip and warn` — Display a warning message in the MATLAB Command Window, and clip as above.

- `Generate error` — Display an error dialog box and terminate the simulation.

When the indexing vector elements are of Boolean data type, the block performs logical indexing. Select **Fill empty spaces in outputs (for logical indexing)** to access the **Fill values** parameter. These values are appended to the output to make it as long as the input elements.

Note The Variable Selector block always copies the selected input rows to a contiguous block of memory (unlike the Simulink Selector block).

Variable Selector

Dialog Box
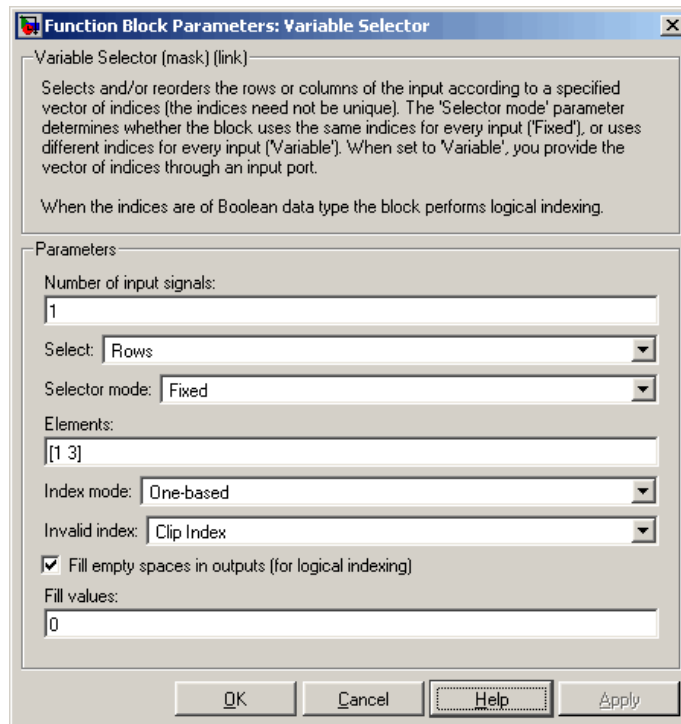
Number of input signals

Specify the number of input signals. An input port is created on the block for each input signal.

Select

The dimension of the input to select, Rows or Columns.

Selector mode

The type of indexing operation to perform, Variable or Fixed. Variable indexing uses the input at the Idx port to select rows or columns from the input at the In port. Fixed indexing uses the **Elements** parameter value to select rows from the input at the In port, and disables the Idx port.

Elements

A vector containing the indices of the input rows or columns that will appear in the output matrix. This parameter is only visible when you select Fixed for the **Selector mode** parameter.

Index mode

When set to One-based, an index value of 1 refers to the first row or column of the input. When set to Zero-based, an index value of 0 refers to the first row or column of the input.

Invalid index

Response to an invalid index value. Tunable.

Fill empty spaces in outputs (for logical indexing)

When the indexing vector elements are of Boolean data type, the block performs logical indexing. This can cause empty spaces in the output. Select this parameter to designate values to be appended to the output in the **Fill values** parameter.

Fill values

Specify the fill values when the block performs logical indexing. This parameter is only visible when the **Fill empty spaces in outputs (for logical indexing)** parameter is selected.

Supported Data Types

Port	Supported Data Types
In	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Custom data types
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

Variable Selector

Port	Supported Data Types
Idx	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Out	• Double-precision floating point • Single-precision floating point • Fixed point (signed and unsigned) • Custom data types • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Multiport Selector	Signal Processing Blockset
Permute Matrix	Signal Processing Blockset
Selector	Simulink
Submatrix	Signal Processing Blockset

Purpose　　　　Compute variance of an input or sequence of inputs

Library　　　　Statistics

Description　　The Variance block computes the variance of each column in the input, or tracks the variance of a sequence of inputs over a period of time. The **Running variance** parameter selects between basic operation and running operation.

Basic Operation

When you do *not* select the **Running variance** check box, the block computes the variance of each column in *M*-by-*N* input matrix u independently at each sample time.

```
y = var(u)      % Equivalent MATLAB code
```

For convenience, length-*M* 1-D vector inputs and *sample-based* length-*M* row vector inputs are both treated as *M*-by-1 column vectors. (A scalar input generates a zero-valued output.)

Running Operation

When you select the **Running variance** check box, the block tracks the variance of each channel in a *time-sequence* of *M*-by-*N* inputs. For sample-based inputs, the output is a sample-based *M*-by-*N* matrix with each element y_{ij} containing the variance of element u_{ij} over all inputs since the last reset. For frame-based inputs, the output is a frame-based *M*-by-*N* matrix with each element y_{ij} containing the variance of the *j*th column over all inputs since the last reset, up to and including element u_{ij} of the current input.

As in basic operation, length-*M* 1-D vector inputs and *sample-based* length-*M* row vector inputs are both treated as *M*-by-1 column vectors.

Resetting the Running Variance

The block resets the running variance whenever a reset event is detected at the optional Rst port. The reset signal rate must be a positive integer multiple of the rate of the data signal input.

Variance

You specify the reset event in the **Reset port** parameter:

- None disables the Rst port.

- Rising edge — Triggers a reset operation when the Rst input does one of the following:

  - Rises from a negative value to a positive value or zero

  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)

- Falling edge — Triggers a reset operation when the Rst input does one of the following:

  - Falls from a positive value to a negative value or zero

  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero (see the following figure)

- Either edge — Triggers a reset operation when the Rst input is a Rising edge or Falling edge (as described above)

- Non-zero sample — Triggers a reset operation at each sample time that the Rst input is not zero

> **Note** When running simulations in the Simulink `MultiTasking`
> mode, reset signals have a one-sample latency. Therefore, when the
> block detects a reset event, there is a one-sample delay at the reset
> port rate before the block applies the reset. For more information on
> latency and the Simulink tasking modes, see "Excess Algorithmic
> Delay (Tasking Latency)" on page 2-56 and the topic on models with
> multiple sample rates in the Real-Time Workshop documentation.

Fixed-Point Data Types

The output at each sample time, y, is a 1-by-*N* vector containing the
variance for each column in u. For purely real or purely imaginary
inputs, the variance of the *j*th column is the square of the standard
deviation:

$$y_j = \sigma_j^2 = \frac{\displaystyle\sum_{i=1}^{M} \left| u_{ij} \right|^2 - \frac{\left| \displaystyle\sum_{i=1}^{M} u_{ij} \right|^2}{M}}{M-1}, 1 \le j \le N$$

For complex inputs, the output is the *total variance* for each column in
u, which is the sum of the real and imaginary variances for that column:

$$\sigma_j^2 = \sigma_{j,Re}^2 + \sigma_{j,Im}^2$$

The following diagram shows the data types used within the Variance
block for fixed-point signals.

Variance

$$u_{ij} \qquad |u_{ij}|^2 \qquad \Sigma|u_{ij}|^2 \qquad \dfrac{\Sigma|u_{ij}|^2 - \dfrac{|\Sigma u_{ij}|^2}{M}}{}$$

Input data type → MAGNITUDE SQUARED → Accumulator or product output data type → ADDER → Accumulator data type

SUBTRACTOR → Accumulator data type

$$u_{ij} \qquad \Sigma u_{ij} \qquad |\Sigma u_{ij}|^2$$

Input data type → ADDER → Accumulator data type → MAGNITUDE SQUARED → Accumulator or product output data type → DIVIDER

Accumulator data type M

DIVIDER → Accumulator data type $\dfrac{|\Sigma u_{ij}|^2}{M}$

$M - 1$

DIVIDER → y_j Output data type

The results of the magnitude squared calculations above are in the product output data type. You can set the accumulator, product output, and output data types in the block dialog as discussed in "Dialog Box" on page 10-1157.

Examples   The Variance block in the model below calculates the running variance of a frame-based 3-by-2 (two-channel) matrix input, u. The running variance is reset at t=2 by an impulse to the block's Rst port.

u [3x2] → In Running / Rst Var [1x2] → yout

Signal From Workspace → Variance → To Workspace

Discrete Impulse
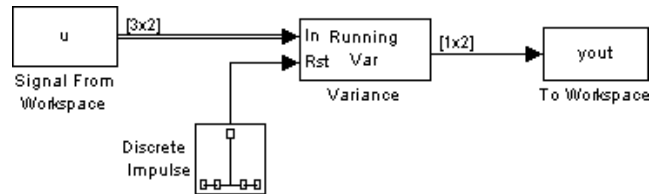
The Variance block has the following settings:

- **Running variance** = ☑
- **Reset port** = Non-zero sample

The Signal From Workspace block has the following settings

- **Signal** = u
- **Sample time** = 1/3
- **Samples per frame** = 3

where

```
u = [6 1 3 -7 2 5 8 0 -1 -3 2 1;1 3 9 2 4 1 6 2 5 0 4 17]'
```

The Discrete Impulse block has the following settings:

- **Delay (samples)** = 2
- **Sample time** = 1
- **Samples per frame** = 1

The block's operation is shown in the figure below.

Variance
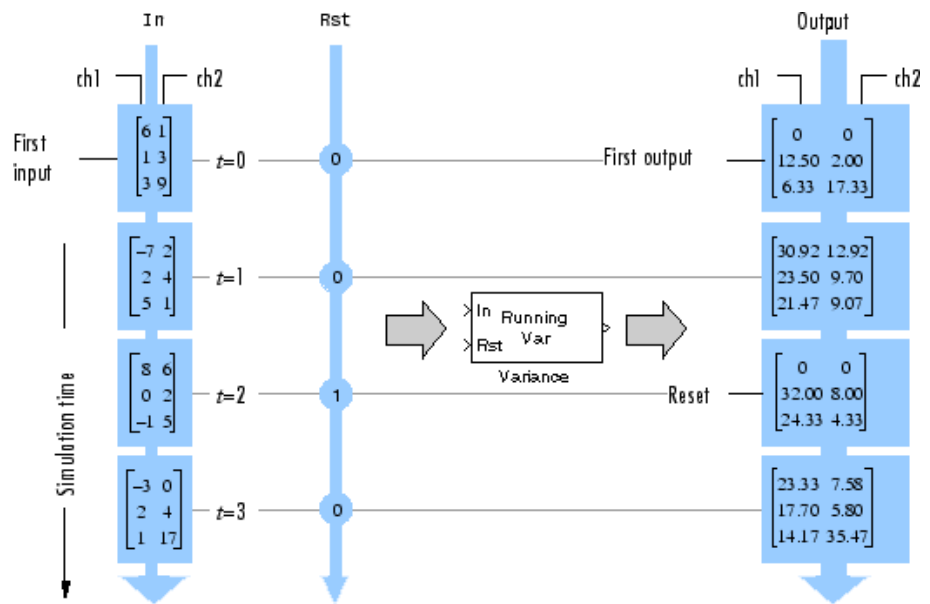
The `statsdem` demo illustrates the operation of several blocks from the Statistics library.

Dialog Box

The **Main** pane of the Variance block dialog appears as follows:
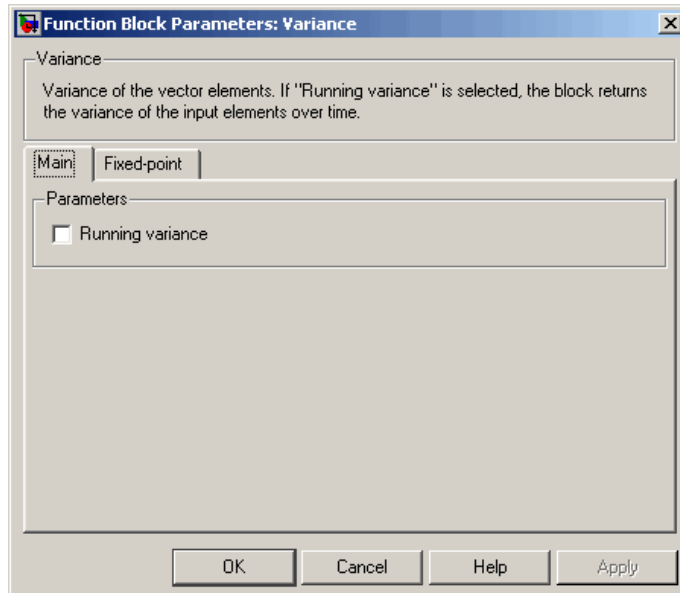
Running variance

Enables running operation when selected.

Reset port

Determines the reset event that causes the block to reset the running variance. The reset signal rate must be a positive integer multiple of the rate of the data signal input. This parameter is enabled only when you select the **Running variance** check box. For more information, see "Resetting the Running Variance" on page 10-1151.

The **Fixed-point** pane of the Variance block dialog appears as follows:

Variance
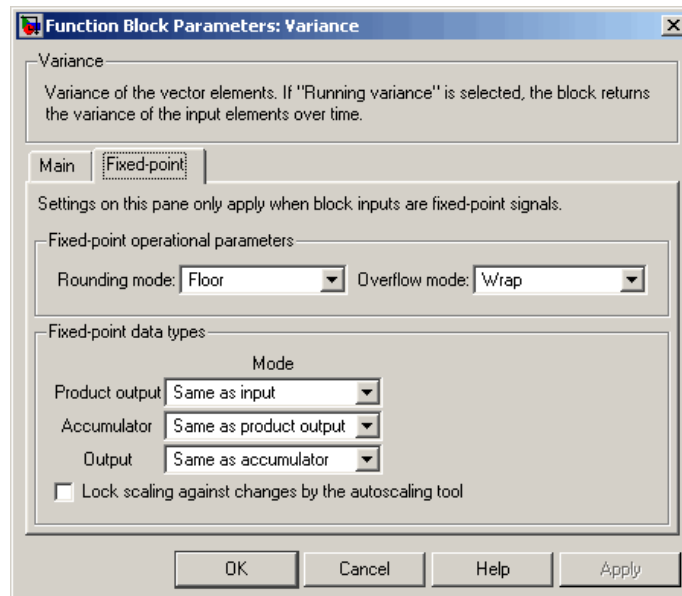
Rounding mode
Select the rounding mode for fixed-point operations.

Overflow mode
Select the overflow mode for fixed-point operations.

> **Note** Refer to "Fixed-Point Data Types" on page 10-1153 for more information on how the product output, accumulator, and output data types are used in this block.

Product output
Use this parameter to specify how you would like to designate the product output word and fraction lengths:

- When you select Same as input, these characteristics will match those of the input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the product output, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the product output. This block requires power-of-two slope and a bias of zero.

Accumulator

Use this parameter to specify the accumulator word and fraction lengths resulting from a complex-complex multiplication in the block:

- When you select `Same as product output`, these characteristics will match those of the product output

- When you select `Same as input`, these characteristics will match those of the input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the accumulator, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the accumulator. This block requires power-of-two slope and a bias of zero.

Output

Choose how you will specify the output word length and fraction length:

- When you select `Same as accumulator`, these characteristics will match those of the accumulator.

- When you select `Same as product output`, these characteristics will match those of the product output.

- When you select `Same as input`, these characteristics will match those of the input to the block.

Variance

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the output, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the output. This block requires power-of-two slope and a bias of zero.

Lock scaling against changes by the autoscaling tool
Select this parameter to prevent any fixed-point scaling you specify in this block mask from being overriden by the autoscaling tool in the Fixed-Point Settings interface. For more information about the autoscaling tool, refer to "Fixed-Point Settings Interface" on page 8-28.

Supported Data Types

- Double-precision floating point

- Single-precision floating point

- Fixed point (signed only)

- Boolean — The block accepts Boolean inputs to the Rst port.

- 8-, 16-, and 32-bit signed integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Mean	Signal Processing Blockset
RMS	Signal Processing Blockset
Standard Deviation	Signal Processing Blockset
var	MATLAB

Purpose   Find vector quantizer codeword that corresponds to a given, zero-based index value

Library   Quantizers

Description
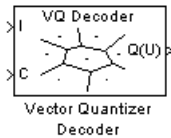
The Vector Quantizer Decoder block associates each input index value with a codeword, a column vector of quantized output values defined in the **Codebook values** parameter. When you input multiple index values into this block, the block outputs a matrix of quantized output vectors. This matrix is created by horizontally concatenating the codeword vectors that correspond to each index value.

You can select how you want to enter the codebook values using the **Source of codebook** parameter. When you select Specify via dialog, you can type the codebook values into the block parameters dialog box. Select Input port and port C appears on the block. The block uses the input to port C as the **Codebook values** parameter.

The **Codebook values** parameter is a *k*-by-*N* matrix of values, where $k \geq 1$ and $N \geq 1$. Each column of this matrix is a codeword vector, and each codeword vector corresponds to an index value. The index values are zero based; therefore, the first codeword vector corresponds to an index value of 0, the second codeword vector corresponds to an index value of 1, and so on.

The input to this block is a vector of index values, where $0 \leq index < N$ and *N* is the number of columns of the codebook matrix. Use the **Action for out of range index value** parameter to determine how the block behaves when an input index value is out of this range. When you want any index values less than 0 to be set to 0 and any index values greater than or equal to *N* to be set to *N*-1, select Clip. When you want to be warned when any index values less than 0 are set to 0 and any index values greater than or equal to *N* are set to *N*-1, select Clip and warn. When you want the simulation to stop and display an error when the index values are out of range, select Error.

Vector Quantizer Decoder

Data Type Support

The input to the block can be the index values and the codebook values. The data type of the index input to the block at port I can be uint8, uint16, uint32, int8, int16, or int32. The data type of the codebook values can be double, single, or Fixed-point.

The output of the block is the quantized output values. These quantized output values always have the same data type as the codebook values. When the codebook values are specified via an input port, the block assigns the same data type to the Q(U) output port. When the codebook values are specified via the dialog, use the **Codebook and output data type** parameter to specify the data type of the Q(U) output port. The data type of the codebook and quantized output can be Same as input, double, single, Fixed-point, or User-defined.

Dialog Box
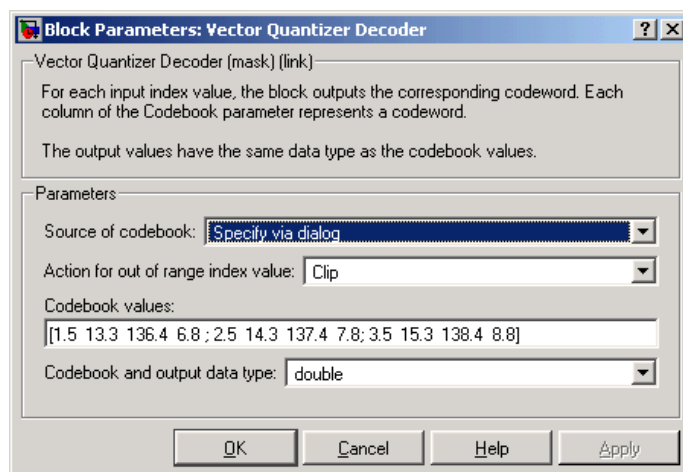
Source of codebook

> Choose Specify via dialog to type the codebook values into the block parameters dialog box. Select Input port to specify the codebook values using the block's input port, C.

Action for out of range index value
Choose the behavior of the block when an input index value is out of range, where $0 \leq index < N$ and *N* is the length of the codebook vector. Select `Clip` when you want any index values less than 0 to be set to 0 and any index values greater than or equal to *N* to be set to *N*-1. Select `Clip and warn` when you want to be warned when any index values less than 0 are set to 0 and any index values greater than or equal to *N* are set to *N*-1. Select `Error` when you want the simulation to stop and display an error when the index values are out of range.

Codebook values
Enter a *k*-by-*N* matrix of quantized output values, where $1 \leq k$ and $1 \leq N$. Each column of your matrix corresponds to an index value. This parameter is visible if, from the **Source of codebook** list, you select `Specify via dialog`.

Codebook and output data type
Use this parameter to specify the data type of the codebook and quantized output values. The data type can be `Same as input`, `double`, `single`, `Fixed-point`, or `User-defined`. This parameter becomes visible when you select `Specify via dialog` for the **Source of codebook** parameter. Nontunable.

Vector Quantizer Decoder
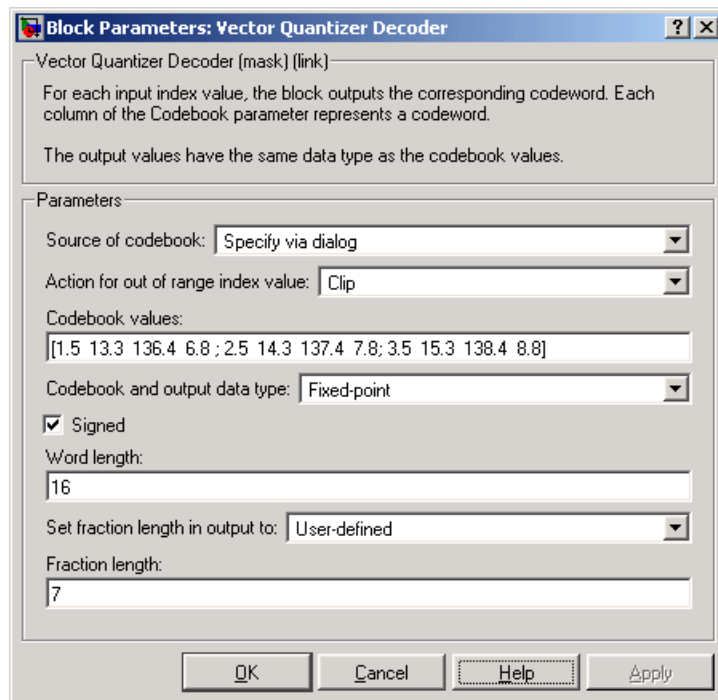
Signed

Select to output a signed fixed-point signal. Otherwise, the signal will be unsigned. This parameter is only visible if, from the **Codebook and output data type** list, you select Fixed-point.

Word length

Specify the word length, in bits, of the fixed-point output data type. This parameter is only visible if, from the **Codebook and output data type** list, you select Fixed-point.

Set fraction length in output to

Specify the scaling of the fixed-point output by either of the following two methods:

- Choose `Best precision` to have the output scaling automatically set such that the output signal has the best possible precision.

- Choose `User-defined` to specify the output scaling in the **Fraction length** parameter.

  This parameter is only visible if, from the **Codebook and output data type** list, you select `Fixed-point` or when you select `User-defined` and the specified output data type is a fixed-point data type.

Fraction length

For fixed-point output data types, specify the number of fractional bits, or bits to the right of the binary point. This parameter is only visible when you select `Fixed-point` or `User-defined` for the **Codebook and output data type** parameter and `User-defined` for the **Set fraction length in output to** parameter.

Vector Quantizer Decoder

User-defined data type

Specify any built-in or fixed-point data type. You can specify fixed-point data types using the `sfix`, `ufix`, `sint`, `uint`, `sfrac`, and `ufrac` functions from Simulink Fixed Point. This parameter is only visible when you select `User-defined` for the **Codebook and output data type** parameter.

References Gersho, A. and R. Gray. *Vector Quantization and Signal Compression*. Boston: Kluwer Academic Publishers, 1992.

Supported Data Types

Port	Supported Data Types
I	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
C	• Double-precision floating point
	• Single-precision floating point
	• Fixed point
	• 8-, 16-, and 32-bit signed integers
Q(U)	• Double-precision floating point
	• Single-precision floating point
	• Fixed point
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Quantizer	Simulink
Scalar Quantizer Decoder	Signal Processing Blockset
Scalar Quantizer Design	Signal Processing Blockset
Uniform Encoder	Signal Processing Blockset
Uniform Decoder	Signal Processing Blockset
Vector Quantizer Encoder	Signal Processing Blockset

Vector Quantizer Design

Purpose   Design vector quantizer using Vector Quantizer Design Tool (VQDTool)

Library   Quantizers
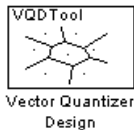
Description Double-click on the Vector Quantizer Design block to start VQDTool, a GUI that allows you to design and implement a vector quantizer. You can also start VQDTool by typing `vqdtool` at the MATLAB command prompt. Based on your specifications, VQDTool iteratively calculates the codebook values that minimize the mean squared error between the training set and the codebook until the stopping criteria for the design process is satisfied. The block uses the resulting codebook values to implement your vector quantizer.

For the **Training Set** parameter, enter a k-by-M matrix of values you want to use to train the quantizer codebook. The variable k, where $k \leq 1$, is the length of each training vector. It also represents the dimension of your quantizer. The variable M, where $M \leq 2$, is the number of training vectors. This data can be created using a MATLAB function, such as the default value `randn(10,1000)`, or it can be any variable defined in the MATLAB workspace.

You have two choices for the **Source of initial codebook** parameter. Select `Auto-generate` to have the block choose the values of the initial codebook. In this case, the block picks N random training vectors as the initial codebook, where N is the **Number of levels** parameter and $N \geq 2$. When you select `User defined`, enter the initial codebook values in the **Initial codebook** field. The initial codebook matrix must have the same number of rows as the training set. Each column of the codebook is a codeword, and your codebook must have at least two codewords.

For the given training set and initial codebook, the block performs an iterative process, using the Generalized Lloyd Algorithm (GLA), to design a final codebook. For each iteration of the GLA, the block first associates each training vector with its nearest codeword by calculating the distortion. You can specify one of the two possible methods for calculating distortion using the **Distortion measure** parameter.

When you select `Squared error` for the **Distortion measure** parameter, the block finds the nearest codeword by calculating the squared error (unweighted). Consider the codebook

$CB = \begin{bmatrix} CW_1 & CW_2 & ... & CW_N \end{bmatrix}$. This codebook has N codewords; each codeword has k elements. The i-th codeword is defined as

$CW_i = \begin{bmatrix} a_{1i} & a_{2i} & ... & a_{ki} \end{bmatrix}$. The training set has M columns and is

defined as $U = \begin{bmatrix} U_1 & U_2 & ... & U_M \end{bmatrix}$, where the p-th training vector

is $U_p = \begin{bmatrix} u_{1p} & u_{2p} & ... & u_{kp} \end{bmatrix}'$. The squared error (unweighted) is calculated using the equation

$$D = \sum_{j=1}^{k} \left(a_{ji} - u_{jp} \right)^2$$

When you select `Weighted squared error` for the **Distortion measure** parameter, enter a vector or matrix for the **Weighting factor** parameter. When the weighting factor is a vector, its length must be equal to the number of rows in the training set. This weighting factor is used for each training vector. When the weighting factor is a matrix, it must be the same size as the training set matrix. The block finds the nearest codeword by calculating the weighted squared error. If

the weighting factor for the p-th column of the training vector, U_p,

is defined as $Wp = \begin{bmatrix} w_{1p} & w_{2p} & ... & w_{kp} \end{bmatrix}'$, then the weighted squared error is defined by the equation

$$D = \sum_{j=1}^{k} w_{jp} \left(a_{ji} - u_{jp} \right)^2$$

Once the block has associated all the training vectors with their nearest codeword vectors, the block calculates the mean squared error for the codebook and checks to see if the stopping criteria for the process has been satisfied.

The two possible options for the **Stopping criteria** parameter are `Relative threshold` and `Maximum iteration`. When you want the design process to stop when the fractional drop in the squared error is below a certain value, select `Relative threshold`. Then, type the maximum acceptable fractional drop in the **Relative threshold** field. The fraction drop in the squared error is defined as

$$\frac{\text{error at previous iteration } - \text{ error at current iteration}}{\text{error at previous iteration}}$$

When you want the design process to stop after a certain number of iterations, choose `Maximum iteration`. Then, enter the maximum number of iterations you want the block to perform in the **Maximum iteration** field. For **Stopping criteria**, you can also choose `Whichever comes first` and enter **Relative threshold** and **Maximum iteration** values. The block stops iterating as soon as one of these conditions is satisfied.

When a training vector has the same distortion for two different codeword vectors, the algorithm uses the **Tie-breaking rule** parameter to determine which codeword vector the training vector is associated with. When you want the training vector to be associated with the lower indexed codeword, select `Lower indexed codeword`. To associate the training vector with the higher indexed codeword, select `Higher indexed codeword`.

With each iteration, the block updates the codeword values in order to minimize the distortion. The **Codebook update method** parameter defines the way the block calculates these new codebook values.

Note If, for the **Distortion measure** parameter, you choose `Squared error`, the **Codebook update method** parameter is set to `Mean`.

If, for the **Distortion measure** parameter, you choose `Weighted squared error` and you choose `Mean` for the **Codebook update method** parameter, the new codeword vector is found as follows.

Suppose there are three training vectors associated with one codeword

vector. The training vectors are $TS_1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$, $TS_3 = \begin{bmatrix} 10 \\ 12 \end{bmatrix}$, and $TS_7 = \begin{bmatrix} 11 \\ 12 \end{bmatrix}$.
The new codeword vector is calculated as

$$CW_{new} = \begin{bmatrix} \dfrac{1+10+11}{3} \\ \dfrac{2+12+12}{3} \end{bmatrix}$$

where the denominator is the number of training vectors associated with this codeword. If, for the **Codebook update method** parameter,

you choose Centroid and you specify the weighting factors $W_1 = \begin{bmatrix} 0.1 \\ 0.2 \end{bmatrix}$,

$W_3 = \begin{bmatrix} 1 \\ 0.6 \end{bmatrix}$, and $W_7 = \begin{bmatrix} 0.3 \\ 0.4 \end{bmatrix}$, the new codeword vector is calculated as

$$CW_{new} = \begin{bmatrix} \dfrac{(0.1)(1)+(1)(10)+(0.3)(11)}{0.1+1+0.3} \\ \dfrac{(0.2)(2)+(0.6)(12)+(0.4)(12)}{0.2+0.6+0.4} \end{bmatrix}$$

Click **Design and Plot** to design the quantizer with the parameter values specified on the left side of the GUI. The performance curve and the entropy of the quantizer are updated and displayed in the figures on the right side of the GUI.

Note You must click **Design and Plot** to apply any changes you make to the parameter values in the VQDTool dialog box.

The following is an example of how the block calculates the entropy of the quantizer at each iteration. Suppose you have a codebook with four

codewords and a training set with 200 training vectors. Also suppose that, at the i-th iteration, 40 training vectors are associated with the first codeword, 60 training vectors are associated with the second codeword, 20 training vectors are associated with the third codeword, and 80 training vectors are associated with the fourth codeword. The probability that a training vector is associated with the first codeword is

$\dfrac{40}{200}$. The probabilities that training vectors are associated with the

second, third, and fourth codewords are $\dfrac{60}{200}$, $\dfrac{20}{200}$, and $\dfrac{80}{200}$,

respectively. The GUI uses these probabilities to calculate the entropy according to the equation

$$H = \sum_{i=1}^{N} -p_i \log_2 p_i$$

where N is the number of codewords. Based on these probabilities, the GUI calculates the entropy of the quantizer at the i-th iteration as

$$H = -\left(\frac{40}{200} \log_2 \frac{40}{200} + \frac{60}{200} \log_2 \frac{60}{200} + \frac{20}{200} \log_2 \frac{20}{200} + \frac{80}{200} \log_2 \frac{80}{200} \right) = 1.8464$$

VQDTool can export parameter values that correspond to the figures displayed in the GUI. Click the **Export Outputs** button, or press **Ctrl+E**, to export the **Final Codebook**, **Mean Square Error**, and **Entropy** values to the workspace, a text file, or a MAT-file.

In the **Model** section of the GUI, specify the destination of the block that will contain the parameters of your quantizer. For **Destination**, select Current model to create a block with your parameters in the model you most recently selected. Type gcs in the MATLAB Command Window to display the name of your current model. Select New model to create a block in a new model file.

From the **Block type** list, select `Encoder` to design a Vector Quantizer Encoder block. Select `Decoder` to design a Vector Quantizer Decoder block. Select `Both` to design a Vector Quantizer Encoder block and a Vector Quantizer Decoder block.

In the **Encoder block name** field, enter a name for the Vector Quantizer Encoder block. In the **Decoder block name** field, enter a name for the Vector Quantizer Decoder block. When you have a Vector Quantizer Encoder and/or Decoder block in your destination model with the same name, select the **Overwrite target block** check box to replace the block's parameters with the current parameters. When you do not select this check box, a new Vector Quantizer Encoder and/or Decoder block is created in your destination model.

Click **Generate Model**. VQDTool uses the parameters that correspond to the current plots to set the parameters of the Vector Quantizer Encoder and/or Decoder blocks.

Vector Quantizer Design

Dialog Box

Training Set

Enter the samples of the signal you would like to quantize. This data set can be a MATLAB function or a variable defined in the MATLAB workspace. The typical length of this data vector is `1e5`.

Source of initial codebook

Select `Auto-generate` to have the block choose the initial codebook values. Choose `User defined` to enter your own initial codebook values.

Number of levels

Enter the number of codeword vectors, *N*, in your codebook matrix, where $N \geq 2$.

Initial codebook

Enter your initial codebook values. From the **Source of initial codebook** list, select `User defined` in order to activate this parameter. The codebook must have the same number of rows as the training set. You must provide at least two codeword vectors.

Distortion measure

When you select `Squared error`, the block finds the nearest codeword by calculating the squared error (unweighted). When you select `Weighted squared error`, the block finds the nearest codeword by calculating the weighted squared error.

Weighting factor

Enter a vector or matrix. The block uses these values to compute the weighted squared error. When the weighting factor is a vector, its length must be equal to the number of rows in the training set. This weighting factor is used for each training vector. When the weighting factor is a matrix, it must be the same size as the training set matrix. The individual weighting factors cannot be negative. The weighting factor vector or matrix cannot contain all zeros.

Stopping criteria

Choose `Relative threshold` to enter the maximum acceptable fractional drop in the squared quantization error. Choose `Maximum iteration` to specify the number of iterations at which to stop.

Choose `Whichever comes first` and the block stops the iteration process as soon as the relative threshold or maximum iteration value is attained.

Relative threshold

This parameter is available when you choose `Relative threshold` or `Whichever comes first` for the **Stopping criteria** parameter. Enter the value that is the maximum acceptable fractional drop in the squared quantization error.

Maximum iteration

This parameter is available when you choose `Maximum iteration` or `Whichever comes first` for the **Stopping criteria** parameter. Enter the maximum number of iterations you want the block to perform.

Tie-breaking rules

When a training vector has the same distortion for two different codeword vectors, select `Lower indexed codeword` to associate the training vector with the lower indexed codeword. Select `Higher indexed codeword` to associate the training vector with the lower indexed codeword.

Codebook update method

When you choose `Mean`, the new codeword vector is calculated by taking the average of all the training vector values that were associated with the original codeword vector. When you choose `Centroid`, the block calculates the new codeword vector by taking the weighted average of all the training vector values that were associated with the original codeword vector Note that if, for the **Distortion measure** parameter, you choose `Squared error`, the **Codebook update method** parameter is set to `Mean`.

Destination

Choose `Current model` to create a Vector Quantizer block in the model you most recently selected. Type `gcs` in the MATLAB Command Window to display the name of your current model. Choose `New model` to create a block in a new model file.

Block type

> Select `Encoder` to design a Vector Quantizer Encoder block. Select `Decoder` to design a Vector Quantizer Decoder block. Select `Both` to design a Vector Quantizer Encoder block and a Vector Quantizer Decoder block.

Encoder block name

> Enter a name for the Vector Quantizer Encoder block.

Decoder block name

> Enter a name for the Vector Quantizer Decoder block.

Overwrite target block

> When you do not select this check box and a Vector Quantizer Encoder and/or Decoder block with the same block name exists in the destination model, a new Vector Quantizer Encoder and/or Decoder block is created in the destination model. When you select this check box and a Vector Quantizer Encoder and/or Decoder block with the same block name exists in the destination model, the parameters of these blocks are overwritten by new parameters.

Generate Model

> Click this button and VQDTool uses the parameters that correspond to the current plots to set the parameters of the Vector Quantizer Encoder and/or Decoder blocks.

Design and Plot

> Click this button to design a quantizer using the parameters on the left side of the GUI and to update the performance curve and entropy plots on the right side of the GUI.
>
> You must click **Design and Plot** to apply any changes you make to the parameter values in the VQDTool GUI.

Export Outputs

> Click this button, or press **Ctrl+E**, to export the **Final Codebook**, **Mean Squared Error**, and **Entropy** values to the workspace, a text file, or a MAT-file.

Vector Quantizer Design

References   Gersho, A. and R. Gray. *Vector Quantization and Signal Compression*. Boston: Kluwer Academic Publishers, 1992.

Supported Data Types

- Double-precision floating point

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Quantizer	Simulink
Scalar Quantizer Decoder	Signal Processing Blockset
Scalar Quantizer Design	Signal Processing Blockset
Uniform Encoder	Signal Processing Blockset
Uniform Decoder	Signal Processing Blockset
Vector Quantizer Decoder	Signal Processing Blockset
Vector Quantizer Encoder	Signal Processing Blockset

Purpose   For a given input, find index of nearest codeword based on Euclidean or weighted Euclidean distance measure

Library   Quantizers

Description
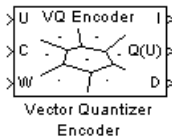
The Vector Quantizer Encoder block compares each input column vector to the codeword vectors in the codebook matrix. Each column of this codebook matrix is a codeword. The block finds the codeword vector nearest to the input column vector and returns its zero-based index. This block supports real floating-point and fixed-point signals on all input ports.

The block finds the nearest codeword by calculating the distortion. The block uses two methods for calculating distortion: Euclidean squared error (unweighted) and weighted Euclidean squared error. Consider the codebook, $CB = \begin{bmatrix} CW_1 & CW_2 & ... & CW_N \end{bmatrix}$. This codebook has N codewords; each codeword has k elements. The i-th codeword is defined as a column vector, $CW_i = \begin{bmatrix} a_{1i} & a_{2i} & ... & a_{ki} \end{bmatrix}$. The multichannel input has M columns and is defined as $U = \begin{bmatrix} U_1 & U_2 & ... & U_M \end{bmatrix}$, where the p-th input column vector is $U_p = \begin{bmatrix} u_{1p} & u_{2p} & ... & u_{kp} \end{bmatrix}'$. The squared error (unweighted) is calculated using the equation

$$D = \sum_{j=1}^{k} \left(a_{ji} - u_{jp} \right)^2$$

The weighted squared error is calculated using the equation

$$D = \sum_{j=1}^{k} w_j \left(a_{ji} - u_{jp} \right)^2$$

where the weighting factor is defined as $W = \begin{bmatrix} w_1 & w_2 & ... & w_k \end{bmatrix}$. The index of the codeword that is associated with the minimum distortion is assigned to the input column vector.

You can select how you want to enter the codebook values using the **Source of codebook** parameter. When you select `Specify via dialog`, you can type the codebook values into the block parameters dialog box. Select `Input port` and port C appears on the block. The block uses the input to port C as the **Codebook** parameter.

The **Codebook** parameter is an *k*-by-*N* matrix of values, where $k \geq 1$ and $N \geq 1$. Each input column vector is compared to this codebook. Each column of the codebook matrix is a codeword, and each codeword has an index value. The first codeword vector corresponds to an index value of 0, the second codeword vector corresponds to an index value of 1, and so on. The codeword vectors must have the same number of rows as the input, U.

For the **Distortion measure** parameter, select `Squared error` when you want the block to calculate the distortion by evaluating the Euclidean distance between the input column vector and each codeword in the codebook. Select `Weighted squared error` when you want to use a weighting factor to emphasize or deemphasize certain input values.

For the **Source of weighting factor** parameter, select `Specify via dialog` to enter a weighting factor vector in the dialog box. Choose `Input port` to specify the weighting factor using port W.

Use the **Weighting factor** parameter to emphasize or deemphasize certain input values when calculating the distortion measure. For example, consider the *p*-th input column vector, U_p, as previously defined. When you want to neglect the effect of the first element of this vector, enter `[0 1 1 ...  1]` as the **Weighting factor** parameter. This weighting factor is used to calculate the weighted squared error using the equation

$$D = \sum_{j=1}^{k} w_j \left(a_{ji} - u_{jp} \right)^2$$

Because of the weighting factor used in this example, the weighted squared error is not affected by the first element of the input matrix.

Therefore, the first element of the input column vector no longer impacts the choice of index value output by the Vector Quantizer Encoder block.

Use the **Index output data type** parameter to specify the data type of the index values output at port I. The data type of the index values can be int8, uint8, int16, uint16, int32, or uint32.

When an input vector is equidistant from two codewords, the block uses the **Tie-breaking rule** parameter to determine which index value the block chooses. When you want the input vector to be represented by the lower index valued codeword, select Choose the lower index. To represent the input column vector by the higher index valued codeword, select Choose the higher index.

Select the **Output codeword** check box to output at port Q(U) the codeword vectors that correspond to each index value. When the input is a matrix, the corresponding codeword vectors are horizontally concatenated into a matrix.

Select the **Output quantization error** check box to output at port *D* the quantization error that results when the block represents the input column vector by its nearest codeword. When the input is a matrix, the quantization error values are horizontally concatenated.

The Vector Quantizer Encoder block accepts real floating-point and fixed-point inputs. For more information on the data types accepted by each port, see "Data Type Support" on page 10-1181 or "Supported Data Types" on page 10-1188.

Data Type Support

The input data values, codebook values, and weighting factor values are input to the block at ports U, C, and W, respectively. The data type of the input data values, codebook values, and weighting factor values can be double, single, or Fixed-point. The input data, codebook values, and weighting factor must be the same data type.

The outputs of the block are the index values, output codewords, and quantization error. Use the **Index output data type** parameter to specify the data type of the index output from the block at port I. The data type of the index can be int8, uint8, int16, uint16, int32, or

uint32. The data type of the output codewords and the quantization error can be double, single, or Fixed-point. The block assigns the data type of the output codewords and the quantization error based on the data type of the input data.

Fixed-Point Data Types

The following diagram shows the data types used within the Vector Quantizer Encoder block for fixed-point signals.
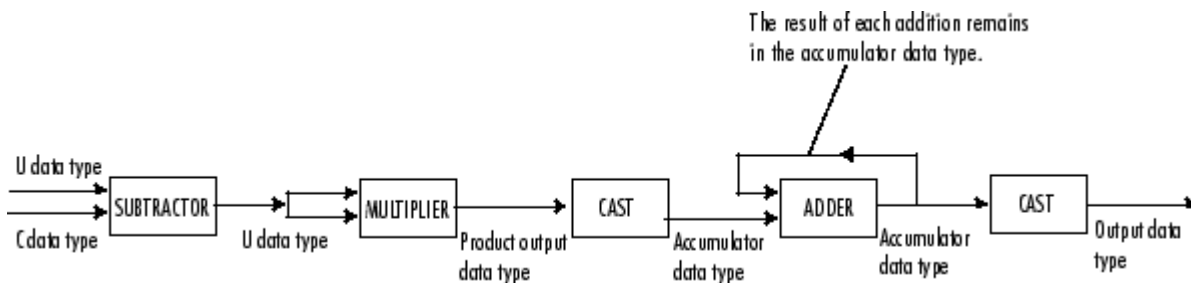
You can set the product output, accumulator, and index output data types in the block dialog as discussed below.

Dialog Box

Block Parameters: Vector Quantizer Encoder

Vector Quantizer Encoder (mask) (link)

For each input column vector, the block outputs a zero-based index value of the nearest codeword. You can choose to output the nearest codeword and corresponding quantization error for each input column vector. Each column of the Codebook parameter represents a codeword. If you choose to specify a weighting factor, it must be a vector having length equal to the number of rows of your input. The block applies the same codebook and weighting factor to each input column vector.

All the inputs to the block must be the same data type. The output index values can be signed or unsigned integers. All other outputs have the same data type as the inputs.

Parameters

Source of codebook: Specify via dialog

Codebook:

[1.5 13.3 136.4 6.8 ; 2.5 14.3 137.4 7.8; 3.5 15.3 138.4 8.8]

Distortion measure: Squared error

Tie-breaking rule: Choose the lower index

☐ Output codeword

☐ Output quantization error

☐ ------------- Show additional parameters -------------

[OK] [Cancel] [Help] [Apply]

Source of codebook

    Choose Specify via dialog to type the codebook values into the block parameters dialog box. Select Input port to specify the codebook values using the block's input port, C.

Codebook

    Enter a k-by-N matrix of values, where $1 \leq k$ and $1 \leq N$, to which your input column vector or matrix is compared. This parameter is visible if, from the **Source of codebook** list, you select Specify via dialog.

Distortion measure

    Select Squared error when you want the block to calculate the distortion by evaluating the Euclidean distance between the input column vector and each codeword in the codebook. Select

Weighted squared error when you want the block to calculate the distortion by evaluating a weighted Euclidean distance using a weighting factor to emphasize or deemphasize certain input values.

Source of weighting factor

Select Specify via dialog to enter a value for the weighting factor in the dialog box. Choose Input port and specify the weighting factor using port W on the block. This parameter is visible if, for the **Distortion measure** parameter, you select Weighted squared error.

Weighting factor

Enter a vector of values. This vector must have length equal to the number of rows of the input, U. This parameter is visible if, for the **Source of weighting factor** parameter, you select Specify via dialog.

Tie-breaking rule

Set this parameter to determine the behavior of the block when an input column vector is equidistant from two codewords. When you want the input column vector to be represented by the lower index valued codeword, select Choose the lower index. To represent the input column vector by the higher index valued codeword, select Choose the higher index.

Output codeword

Select this check box to output the codeword vectors nearest to the input column vectors.

Output quantization error

Select this check box to output the quantization error value that results when the block represents the input column vector by the nearest codeword.
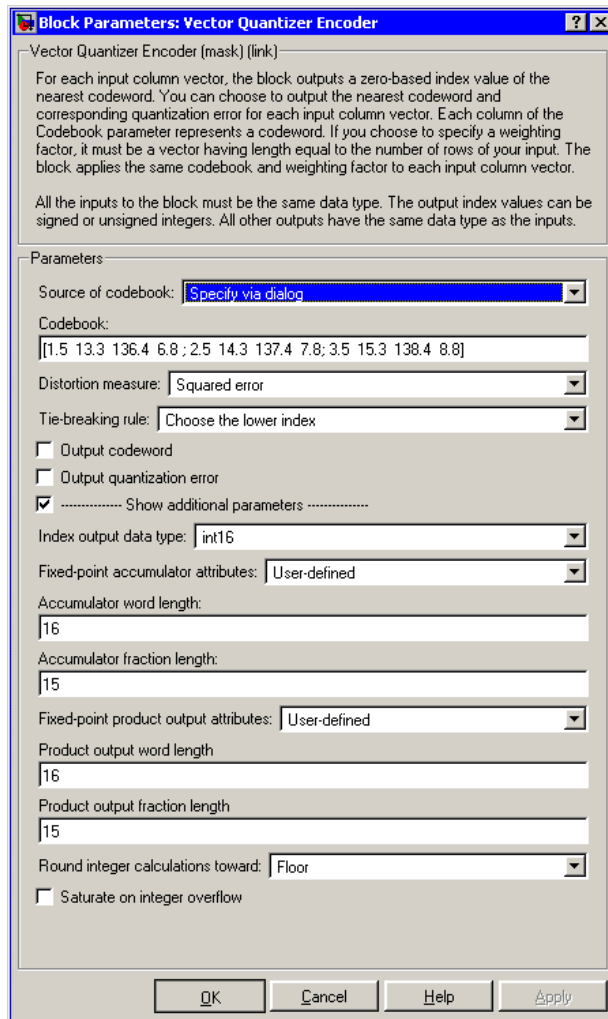
Index output data type

Select int8, uint8, int16, uint16, int32, or uint32 as the data type of the index output at port I.
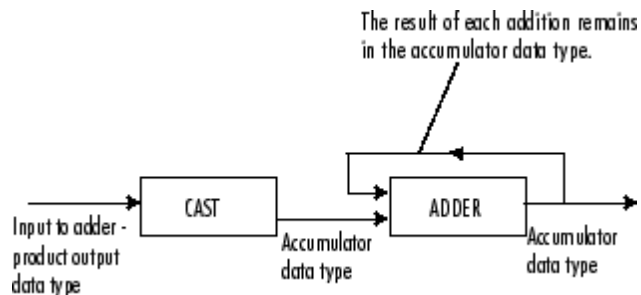
Fixed-point accumulator attributes

The result of each addition remains
in the accumulator data type.

As depicted above, inputs to the accumulator are cast to the accumulator data type. The output of the adder remains in the accumulator data type as each element of the input is added to it. Use this parameter to specify how you would like to designate this accumulator word and fraction lengths.

When you select `Same as input`, the accumulator word and fraction lengths are the same as those of the input of the block. When you select `User-defined`, the **Accumulator word length** and **Accumulator fraction length** parameters become visible.

Accumulator word length
Specify the word length, in bits, of the accumulator. This parameter is only visible when you specify `User-defined` for the **Fixed-point accumulator attributes** parameter.

Accumulator fraction length
Specify the fraction length, in bits, of the accumulator. This parameter is only visible when you specify `User-defined` for the **Fixed-point accumulator attributes** parameter.

Fixed-point product output attributes
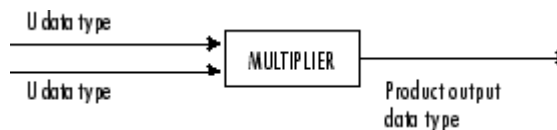
As depicted above, the output of the multiplier is placed into the product output data type and scaling. Use this parameter to specify how you would like to designate this product output word and fraction lengths.

When you select Same as accumulator, the product output word and fraction lengths are the same as those of the accumulator of the block. When you select Same as input, they are the same as those of the input of the block. When you select User-defined, the **Product output word length** and **Product output fraction length** parameters become visible.

Product output word length
Specify the word length, in bits, of the product output. This parameter is only visible when you specify User-defined for the **Fixed-point product output attributes** parameter.

Product output fraction length
Specify the fraction length, in bits, of the product output. This parameter is only visible when you specify User-defined for the **Fixed-point product output attributes** parameter.

Round integer calculations toward
Select the rounding mode for fixed-point operations. This parameter is visible when you select the **Show additional parameters** check box.

Saturate on integer overflow
When selected, overflows saturate. This parameter is visible when you select the **Show additional parameters** check box.

References Gersho, A. and R. Gray. *Vector Quantization and Signal Compression*. Boston: Kluwer Academic Publishers, 1992.

Vector Quantizer Encoder

Port	Supported Data Types
U	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
C	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
W	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
I	• 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Q(U)	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers
D	• Double-precision floating point • Single-precision floating point • Fixed point (signed only) • 8-, 16-, and 32-bit signed integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Quantizer	Simulink
Scalar Quantizer Decoder	Signal Processing Blockset
Scalar Quantizer Design	Signal Processing Blockset
Uniform Encoder	Signal Processing Blockset
Uniform Decoder	Signal Processing Blockset
Vector Quantizer Decoder	Signal Processing Blockset

Vector Scope

Purpose   Display vector or matrix of time-domain, frequency-domain, or user-defined data

Library   Signal Processing Sinks

Description   The Vector Scope block is a comprehensive display tool similar to a digital oscilloscope. The block can display time-domain, frequency-domain, or user-defined signals. You can use the Vector Scope block to plot consecutive time samples from a frame-based vector, or to plot vectors containing data such as filter coefficients or spectral magnitudes. To compute and plot the periodogram of a signal with a single block, use the Spectrum Scope block.

The input to the Vector Scope block can be any real-valued M-by-N matrix, column or row vector, or 1-D (unoriented) vector, where 1-D vectors are treated as column vectors. Regardless of the input frame status, the block treats each column of an M-by-N input as an independent channel of data with M consecutive samples.

The block plots each sample of each input channel sequentially across the horizontal axis of the plot.

Scope Properties Pane

The **Scope Properties** pane enables you to plot time-domain, frequency-domain, or user-defined data, and adjust the horizontal display span of the plot. The scope displays frames of data, and updates the display for each new input frame.

The **Input domain** parameter specifies the domain of the input data. If you select Time, for M-by-N inputs containing time-domain data, the block treats each of the N input frames (columns) as a succession of M consecutive samples taken from a time series. That is, each data point in the input frame is assumed to correspond to a unique time value.

If, for the **Input domain** parameter, you select Frequency, for M-by-N inputs containing frequency-domain data, the block treats each of the N input frames (columns) as a vector of spectral magnitude data corresponding to M consecutive ascending frequency indices. That is, when the input is a single column vector, u, each value in the input

frame, `u(i)`, is assumed to correspond to a unique frequency value, `f(i)`, where `f(i+1)>f(i)`.

If, for the **Input domain** parameter, you select `User-defined`, the block does not assume that the input frame data is time-domain or frequency-domain data. You can plot the data in the appropriate manner. Also, the **Horizontal display span (number of frames)** parameter appears on the pane. Enter a scalar value greater than or equal to one that corresponds to the number of frames to be displayed across the width of the scope window.
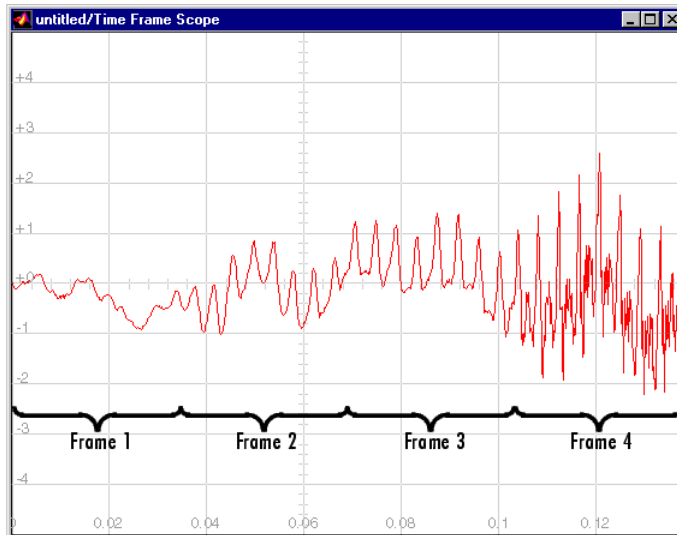
If, for the **Input domain** parameter you choose `Time`, the **Time display span (number of frames)** parameter appears on the pane. Enter a scalar value greater than or equal to one that corresponds to the number of frames to be displayed across the width of the scope window.

Time-Domain Scaling

The block scales the horizontal (time) axis of time-domain signals automatically. The range of the time axis is $[0,S*T_{fi}]$, where T_{fi} is the input frame period, and S is the **Time display span (number of**

frames) parameter. The spacing between time points is $T_{fi}/(M\text{-}1)$, where M is the number of samples in each consecutive input frame. Frequency-domain and user-defined data need additional information to scale the horizontal axis. For more information, see "Frequency-Domain Scaling" on page 10-1195 and "User-Defined Domain Scaling" on page 10-1196.

Display Properties Pane

The **Display Properties** pane enables you to control how the block displays your data.

The **Show grid** parameter toggles the background grid on and off.

If you select the **Persistence** check box, the window maintains successive displays. That is, the scope does not erase the display after each frame (or collection of frames), but overlays successive input frames in the scope display.

If you select the **Frame number** check box, the block displays the number of the current frame in the input sequence on the scope window, and the block increments the count as each new input is received. Counting starts at 1 with the first input frame, and continues until the simulation stops.

If you select the **Channel legend** check box, a legend indicating the line color, style, and marker of each channel's data is added. When the input signal is labeled, that label is displayed in the channel legend. When the input signal is not labeled, but comes from a Matrix Concatenation block with labeled inputs, those labels are displayed in the channel legend. Otherwise, each channel in the legend is labeled with the channel number (CH 1, CH 2, etc.). Click-and-drag the legend to reposition it in the scope window; double-click on the line label to edit the text. If you rerun the simulation, the labels revert to the defaults.

If you select the **Compact display** check box, the scope completely fills the figure window. The scope does not display menus and axis titles, and it shows the numerical axis labels within the axes. If you clear the **Compact display** check box, the scope displays the axis labels and

titles in a gray border surrounding the scope axes, and the window's menus and toolbar are visible.

If you select the **Open scope at start of simulation** check box, the scope opens at the start of the simulation. If you clear this parameter, the scope does not open automatically during the simulation. To view the scope, double-click the Vector Scope block, which brings up the scope as well as the block parameter dialog box. Use this feature when you have several scope blocks in a model and you do not want to view all the associated scopes during the simulation.

If the scope is not open during the simulation and you select the **Open scope immediately** check box, the block opens the scope and clears the check box.

The **Scope position** parameter specifies a four-element vector of the form

```
[left bottom width height]
```

specifying the position of the scope window on the screen, where (0,0) is the lower-left corner of the display. See the MATLAB `figure` function for more information.

Axis Properties Pane

The parameters on the **Axis Properties** pane vary based on the value of the **Input domain** parameter on the **Scope Properties** pane.

The following text describes the parameters available for time domain inputs.

Minimum Y-limit and **Maximum Y-limit** parameters set the range of the vertical axis.

The **Y-axis title** is the text to be displayed to the left of the y-axis.

The following text describes the parameters available for frequency domain inputs.

The **Frequency units** parameter specifies whether the frequency axis values should be in units of Hertz or rad/sec, When the **Frequency**

units parameter is set to `Hertz`, the spacing between frequency points is $1/(M*T_s)$, where T_s is the sample time of the original time-domain signal. When the **Frequency units** parameter is set to `rad/sec`, the spacing between frequency points is $2\pi/(M*T_s)$.

The **Frequency range** parameter specifies the range of frequencies over which the magnitudes in the input should be plotted. The available options are `[0..Fs/2]`, `[-Fs/2..Fs/2]`, and `[0..Fs]`, where `Fs` is the original time-domain signal's sample frequency. The Vector Scope block assumes that the input data spans the range $[0,F_s)$, which is the same as the output from an FFT. To plot over the range `[0..Fs/2]` the scope truncates the input vector leaving only the first half of the data, then plots these remaining samples over half the frequency range. To plot over the range `[-Fs/2..Fs/2]`, the scope reorders the input vector elements such that the last half of the data becomes the first half, and vice versa; then it relabels the *x*-axis accordingly.

If, for frequency domain inputs, you select the **Inherit sample time from input** check box, the block scales the frequency axis by reconstructing the frequency data from the frame-period of the frequency-domain input. This is valid when the following conditions hold:

- Each frame of frequency-domain data shares the same length as the frame of time-domain data from which it was generated; for example, when the FFT is computed on the same number of points as are contained in the time-domain input.

- The sample period of the time-domain signal in the simulation is equal to the period with which the physical signal was originally sampled.

- Consecutive frames containing the time-domain signal do not overlap each other; that is, a particular signal sample does not appear in more than one sequential frame.

In cases where not all of these conditions hold, you should specify the appropriate value for the **Sample time of original time series** parameter.

The **Amplitude scaling** parameter allows you to select Magnitude or dB scaling along the *y*-axis.

Minimum Y-limit and **Maximum Y-limit** parameters set the range of the vertical axis.

The **Y-axis title** is the text to be displayed to the left of the y-axis.

Frequency-Domain Scaling

To correctly scale the horizontal (frequency) axis for frequency-domain signals, the Vector Scope block needs to know the sample period of the original time-domain sequence represented by the frequency-domain data. You specify this period by entering a value for the **Sample time of original time series** parameter. For additional information, see "Time-Domain Scaling" on page 10-1191 and "User-Defined Domain Scaling" on page 10-1196.

The following text describes the parameters available for user-defined domain inputs.

If, for user-defined input domains, you select the **Inherit sample increment from input** check box, the block scales the horizontal axis by computing the horizontal interval between samples in the input frame from the frame period of the input. For example, when the input frame period is 1, and there are 64 samples per input frame, the interval between samples is computed to be 1/64. Computing the interval this way is usually only valid when the following conditions hold:

- The input is a nonoverlapping time series; the *x*-axis on the scope represents time.

- The input's sample period (1/64 in the above example) is equal to the period with which the physical signal was originally sampled.

In cases where not all of these conditions hold, you should use the **Increment per sample in input frame** parameter.

The **Scope position** parameter specifies a four-element vector of the form

```
[left bottom width height]
```

specifying the position of the scope window on the screen, where (0,0) is the lower-left corner of the display. See the MATLAB `figure` function for more information.

Minimum Y-limit and **Maximum Y-limit** parameters set the range of the vertical axis.

The **Y-axis title** is the text to be displayed to the left of the y-axis.

User-Defined Domain Scaling

To correctly scale the horizontal axis for user-defined input domains, the block needs to know the spacing of the input data. You specify this spacing using the **Increment per sample in input frame** parameter, I_s. This parameter represents the numerical interval between adjacent x-axis points corresponding to the input data. The range of the horizontal axis is $[0, M*I_s*S]$, where M is the number of samples in each consecutive input frame, and S is the **Horizontal display span (number of frames) parameter** you specify in the **Scope Properties** pane. For additional information, see "Time-Domain Scaling" on page 10-1191 and "Frequency-Domain Scaling" on page 10-1195.

Line Properties Pane

Use the parameters on the **Line Properties** pane to help you distinguish between two or more independent channels of data on the scope.

The **Line visibilities** parameter specifies which channel's data is displayed on the scope, and which is hidden. The syntax specifies the visibilities in list form, where the term `on` or `off` as a list entry specifies the visibility of the corresponding channel's data. The list entries are separated by the pipe symbol, `|`.

For example, a five-channel signal would ordinarily generate five distinct plots on the scope. To disable plotting of the third and fifth lines, enter the following visibility specification in the **Line visibilities** parameter.

```
 on | on | off | on | off
ch 1  ch 2  ch 3  ch 4  ch 5
```

Note that the first (leftmost) list item corresponds to the first signal channel (leftmost column of the input matrix).

The **Line styles** parameter specifies the line style with which each channel's data is displayed on the scope. The syntax specifies the channel line styles in list form, with each list entry specifying a style for the corresponding channel's data. The list entries are separated by the pipe symbol, |.

For example, a five-channel signal would ordinarily generate all five plots with a solid line style. To plot each line with a different style, enter

```
  - | -- | : | -. | -
ch 1 ch 2 ch 3 ch 4  ch 5
```

These settings plot the signal channels with the following styles.

Line Style	Command to Type in Line Style Parameter	Appearance
Solid	-	————————————
Dashed	- -	— — — — — — —
Dotted	:	································
Dash-dot	-.	— · — · — · — · — · — ·
No line	none	No line appears

Note that the first (leftmost) list item, `'-'`, corresponds to the first signal channel (leftmost column of the input matrix). See the `LineStyle`
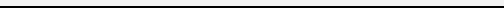
property of the MATLAB `line` function for more information about the style syntax.

The **Line markers** parameter specifies the marker style with which each channel's samples are represented on the scope. The syntax specifies the channels' marker styles in list form, with each list entry specifying a marker for the corresponding channel's data. The list entries are separated by the pipe symbol, |.

For example, a five-channel signal would ordinarily generate all five plots with no marker symbol (that is, the individual sample points are not marked on the scope). To instead plot each line with a different marker style, you could enter

```
    *  |  .  |  x  |  s  |  d
ch 1   ch 2   ch 3   ch 4   ch 5
```

These settings plot the signal channels with the following styles.

Marker Style	Command to Type in Marker Style Parameter	Appearance
Asterisk	*	
Point	.	
Cross	x	
Square	s	
Diamond	d	

Note that the leftmost list item, `'*'`, corresponds to the first signal channel or leftmost column of the input matrix. See the `Marker` property of the MATLAB `line` function for more information about the available markers.

To produce a *stem plot* for the data in a particular channel, type the word `stem` instead of one of the basic marker shapes.

The **Line colors** parameter specifies the color in which each channel's data is displayed on the scope. The syntax specifies the channel colors in list form, with each list entry specifying a color (in one of the MATLAB `ColorSpec` formats) for the corresponding channel's data. The list entries are separated by the pipe symbol, `|`.
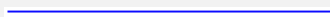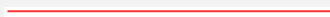
For example, a five-channel signal would ordinarily generate all five plots in the color black. To instead plot the lines with the color order below, enter

```
[0 0 0] | [0 0 1] | [1 0 0 ] | [0 1 0] | [.7529 0 .7529]
 ch 1       ch 2        ch 3       ch 4        ch 5
```

or

```
'k' | 'b' | 'r' | 'g' | [.7529 0 .7529]
ch 1   ch 2   ch 3   ch 4       ch 5
```

These settings plot the signal channels in the following colors (8-bit RGB equivalents shown in the center column).

Color	RGB Equivalent	Appearance
Black	(0,0,0)	
Blue	(0,0,255)	
Red	(255,0,0)	

Color	RGB Equivalent	Appearance
Green	(0,255,0)	
Dark purple	(192,0,192)	

Note that the leftmost list item, `'k'`, corresponds to the first signal channel or leftmost column of the input matrix. See the MATLAB function `ColorSpec` for more information about the color syntax.

Vector Scope Window

The title in the window title bar is the same as the block title. In addition to the standard MATLAB figure window menus such as **File**, **Window**, and **Help**, the Vector Scope window contains **Axes** and **Channels** menus.

The parameters that you set using the **Axes** menu apply to all channels. Many of the parameters in this menu are also accessible through the block parameters dialog box. For descriptions of these parameters, see "Display Properties Pane" on page 10-1192. Below are descriptions of other parameters in the **Axes** menu:

- **Refresh** erases all data on the scope display, except for the most recent trace. This command is useful in conjunction with the **Persistence** setting.

- **Autoscale** resizes the *y*-axis to best fit the vertical range of the data. The numerical limits selected by the autoscale feature are displayed in the **Minimum Y-limit** and **Maximum Y-limit** parameters in the parameter dialog box. You can edit these values.

- **Save position** automatically updates the **Scope position** parameter in the **Axis properties** field to reflect the scope window's current position and size. To make the scope window open at a particular location on the screen when the simulation runs, drag the window to the desired location, resize it, and select **Save position**. Note

that the parameter dialog box must be closed when you select **Save position** in order for the **Scope position** parameter to be updated.

The properties listed in the **Channels** menu apply to a particular channel. All of the parameters in this menu are also accessible through the block parameters dialog box. For descriptions of these parameters, see "Line Properties Pane" on page 10-1196.

Many of these options can also be accessed by right-clicking with the mouse anywhere on the scope display. The menu that is displayed contains a combination of the options available in both the **Axes** and **Channels** menus.

Note When you select **Compact display** from the **Axes** menu, the **Axes** and **Channels** menus are no longer visible. Right-click in the Vector Scope window and click **Compact display** in order to make the menus reappear.

Vector Scope

Dialog Box

Scope Properties Pane
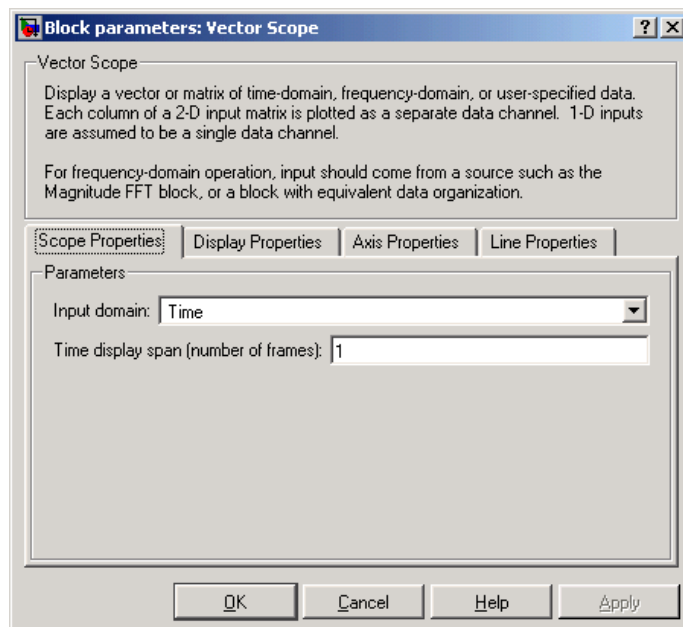
Input domain

Select the domain of the input. Your choices are Time, Frequency, or User-defined. Tunable.

Time display span (number of frames)

The number of consecutive frames to display (horizontally) on the scope at any one time. This parameter is visible when the **Input domain** parameter is set to Time.

Horizontal display span (number of frames)

The number of consecutive frames to display (horizontally) on the scope at any one time. This parameter is visible when the **Input domain** parameter is set to User-defined.

Display Properties Pane
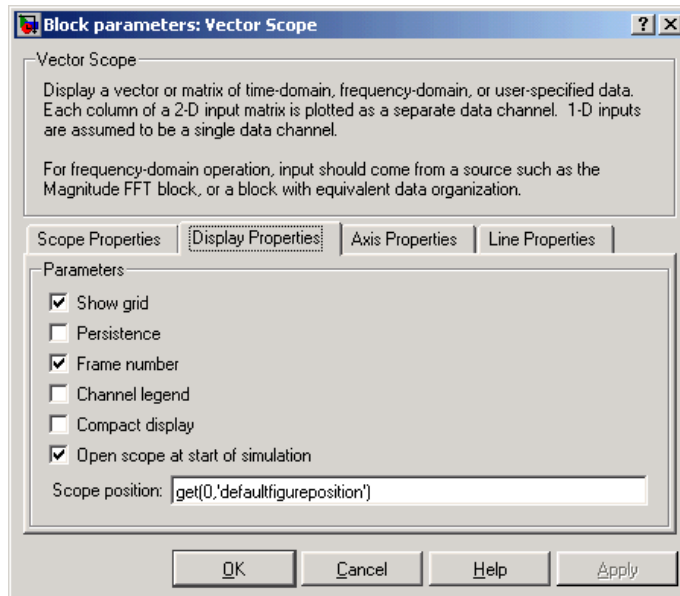
Show grid

    Toggle the scope grid on and off. Tunable.

Persistence

    Select this check box to maintain successive displays. That is, the scope does not erase the display after each frame (or collection of frames), but overlays successive input frames in the scope display. Tunable.

Frame number

    If you select this check box, the number of the current frame in the input sequence appears in the Vector Scope window. Tunable.

Channel legend

    Toggles the legend on and off. Tunable.

Vector Scope

Compact display

Resizes the scope to fill the window. Tunable.

Open scope at start of simulation

Select this check box to open the scope at the start of the simulation. When this parameter is cleared, the scope will not open automatically during the simulation. Tunable.

Open scope immediately

If the scope is not open during simulation, select this check box to open it. This parameter is visible only while the simulation is running.

Scope position

A four-element vector of the form `[left bottom width height]` specifying the position of the scope window. `(0,0)` is the lower-left corner of the display. Tunable.

Axis Properties Pane
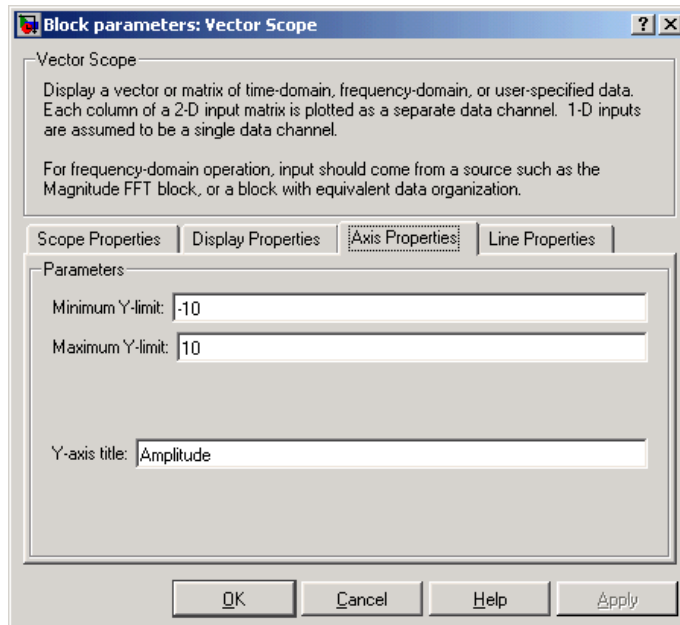
Minimum Y-limit
>   The minimum value of the *y*-axis. Tunable.

Maximum Y-limit
>   The maximum value of the *y*-axis. Tunable.

Y-axis title
>   The text to be displayed to the left of the *y*-axis. Tunable.

Frequency units
>   Choose the frequency units for the *x*-axis, Hertz or rad/sec. This parameter is visible when, in the **Scope Properties** pane, for the **Input domain** parameter, you select Frequency. Tunable.

Frequency range

Specify the frequency range over which to plot the data. This parameter is visible when, in the **Scope Properties** pane, for **Input domain** parameter, you select `Frequency`. Tunable.

Inherit sample time from input

If you select this check box, the block computes the time-domain sample period from the frame period and frame size of the frequency-domain input. Use this parameter only when the length of the each frame of frequency-domain data is the same as the length of the frame of time-domain data from which it was generated. This parameter is visible when, in the **Scope Properties** pane, for **Input domain** parameter, you select `Frequency`. Tunable.

Sample time of original time series

Enter the sample period, T_s, of the original time-domain signal. This parameter is available when, in the **Scope Properties** pane, for **Input domain** parameter, you select `Frequency`. Then, in the **Axis Properties** pane, you clear the **Inherit sample time from input** check box. Tunable.

Amplitude scaling

Choose the scaling for the *y*-axis, `dB` or `Magnitude`. This parameter is visible when, in the **Scope Properties** pane, for **Input domain** parameter, you select `Frequency`. Tunable.

Inherit sample increment from input

If you select this check box, the block scales the horizontal axis by computing the horizontal interval between samples in the input frame from the frame period of the input. Use this parameter only when the input's sample period is equal to the period with which the physical signal was originally sampled. This parameter is visible when, in the **Scope Properties** pane, for **Input domain** parameter, you select `User-defined`. Tunable.

Increment per sample in input frame

Enter the numerical interval between adjacent *x*-axis points corresponding to the user-defined input data. This parameter

is available when, in the **Scope properties** pane, for **Input domain** parameter, you select User-defined. Then, in the **Axis Properties** pane you clear the **Inherit sample increment from input** check box. Tunable.

X-axis title

Enter the text to be displayed below the *x*-axis. This parameter is visible when, in the **Scope properties** pane, for **Input domain** parameter, you select User-defined. Tunable.

Line Properties Pane
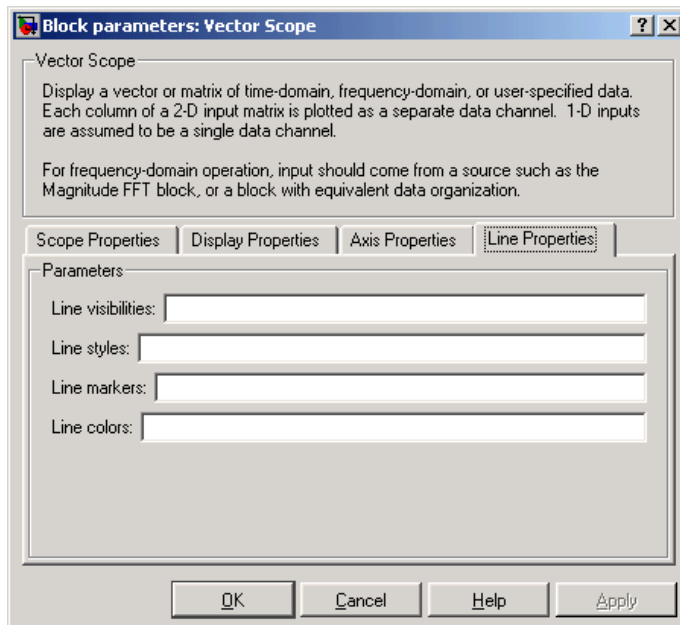
Line visibilities

Enter on or off to specify the visibility of the various channels' scope traces. Separate your choices for each channel with by a pipe (|) symbol. Tunable.

Vector Scope

Line styles

Enter the line styles of the various channels' scope traces. Separate your choices for each channel with by a pipe (|) symbol. Tunable.

Line markers

Enter the line markers of the various channels' scope traces. Separate your choices for each channel with by a pipe (|) symbol. Tunable.

Line colors

Enter the colors of the various channels' scope traces using the ColorSpec formats. Separate your choices for each channel with by a pipe (|) symbol. Tunable.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• Custom data types
	• Boolean
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

| Matrix Viewer | Signal Processing Blockset |
| Spectrum Scope | Signal Processing Blockset |

Waterfall

Purpose   View vectors of data over time

Library   Signal Processing Sinks

Description   The Waterfall block displays multiple vectors of data at one time. These vectors represent the input data at consecutive sample times. The input to the block can be real or complex-valued data vectors of any data type including fixed-point data types. However, the input is converted to double-precision before the block processes the data. The Waterfall block displays only real-valued, double-precision vectors of data.

The data is displayed in a three-dimensional axis in the Waterfall window. By default, the *x*-axis represents amplitude, the *y*-axis represents samples, and the *z*-axis represents time. You can adjust the number of sample vectors that the block displays, move and resize the Waterfall window, and modify block parameter values during the simulation. The Waterfall window has toolbar buttons that enable you to zoom in on displayed data, suspend data capture, freeze the scope's display, save the scope position, and export data to the workspace. The toolbar buttons are labeled in the following figure, which shows the Waterfall window as it appears when you double-click a Waterfall block.

Waterfall
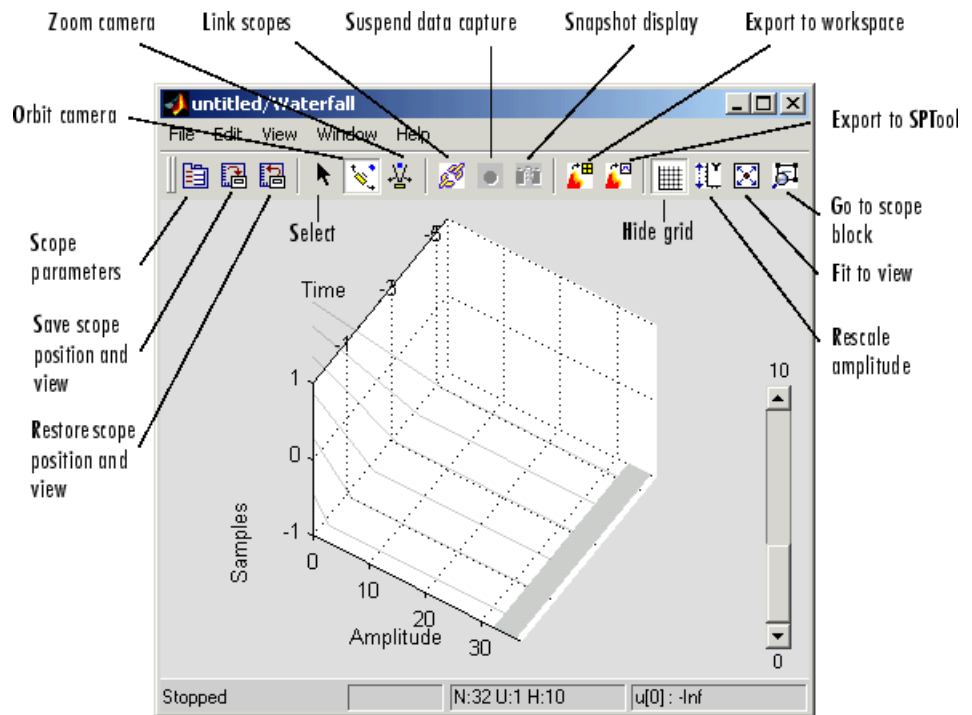
Sections of This Reference Page

- "Waterfall Parameters" on page 10-1211
- "Display Parameters" on page 10-1212
- "Axes Parameters" on page 10-1213
- "Data History Parameters" on page 10-1214
- "Triggering Parameters" on page 10-1215
- "Scope Trigger Function" on page 10-1218
- "Transform Parameters" on page 10-1221
- "Scope Transform Function" on page 10-1223

• "Examples" on page 10-1223

Waterfall Parameters

You can control the display and behavior of the Waterfall window using the Parameters dialog box.

Note You can alter the Waterfall parameters while the simulation is running. However, when you make changes to values in text boxes, you must click **Enter** or click outside the text box before the block accepts your changes.

---

1 To open the Parameters dialog box, click the **Scope parameters** button.
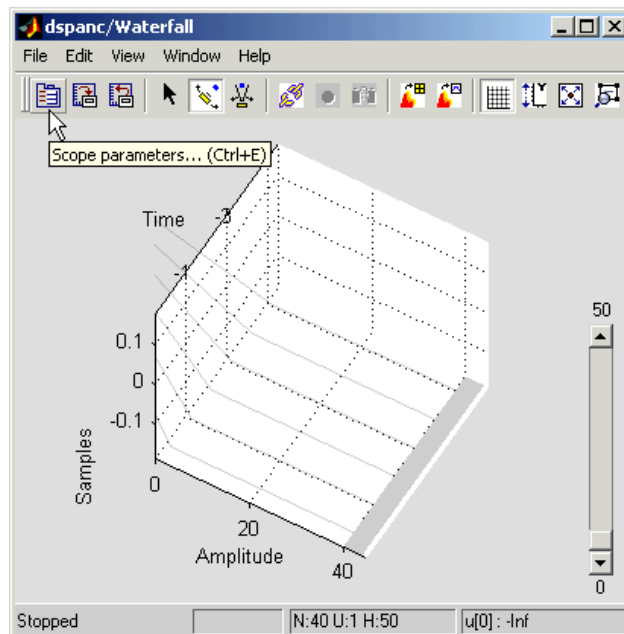
Waterfall

The Parameters dialog box appears.
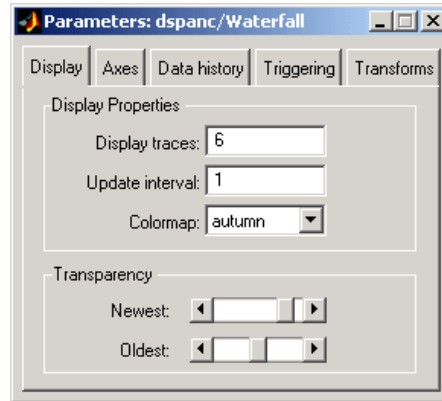
2 Click on the different panes to enter parameter settings.

Display Parameters

The following parameters control the Waterfall window's display.
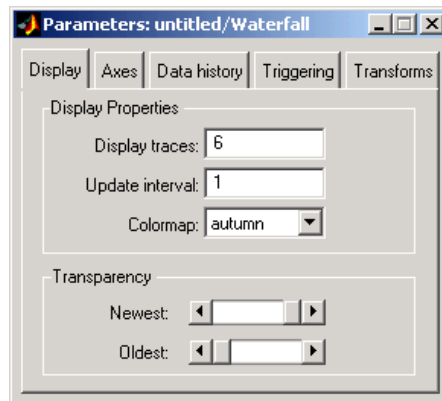
Display traces

Enter the number of vectors of data to be displayed in the Waterfall window.

Update interval

Enter the number of vectors the block should store before it displays them to the window.

Colormap

Choose a colormap for the displayed data.

Transparency

Specify the transparency of the newest and oldest data vectors. Placing the slider in the left-most position tells the block to make the data vector transparent. Placing the slider in the right-most position tells the block to make the data vector opaque. The intermediate data vectors transition between the two chosen transparency values.

Axes Parameters

The following parameters control the axes in the Waterfall window.
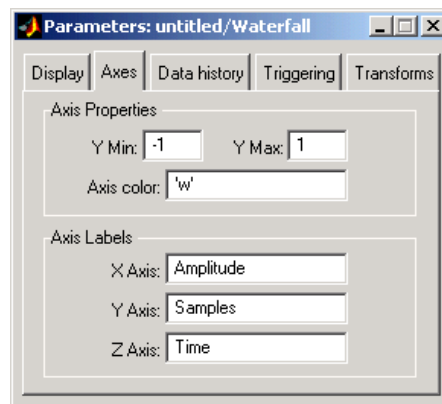
Y Min

Enter the minimum value of the *y*-axis.

Y Max

Enter the maximum value of the *y*-axis.

Axis color

Enter a background color for the axes. Specify the color using a character string. For example, to specify black, enter `'k'`.

X Axis

Enter the *x*-axis label.

Y Axis

Enter the *y*-axis label.

Z Axis

Enter the *z*-axis label.

Data History Parameters

The following parameters control how many input data vectors the Waterfall block stores. They also control how the data is exported to the MATLAB workspace or SPTool.
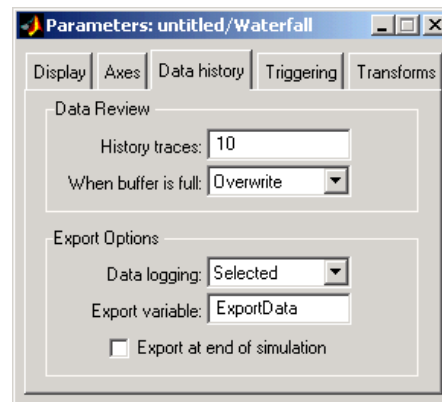
History traces

Enter the number of vectors (traces) that you want the block to store.

When the buffer is full

Use this parameter to control the behavior of the block when the buffer is filled:

- Overwrite — The old data is replaced with the new data.

- Suspend — The block stops storing data in the buffer; however, the simulation continues to run.

- Extend — The block extends the buffer so that it can continue to store all the input data.

Data logging

Use this parameter to control which data is exported from the block:

- Selected — The selected data vector is exported.

- All visible — All of the data vectors displayed in the Waterfall window are exported.

- All history — All of the data vectors stored in the block's history buffer are exported.

Export variable

Enter the name of the variable that represents your data in the MATLAB workspace or SPTool. The default variable name is ExportData.

Export at end of simulation

Select this check box to automatically export the data to the MATLAB workspace when the simulation stops.

Triggering Parameters

The following parameters control when the Waterfall block starts and stops capturing data.

Waterfall
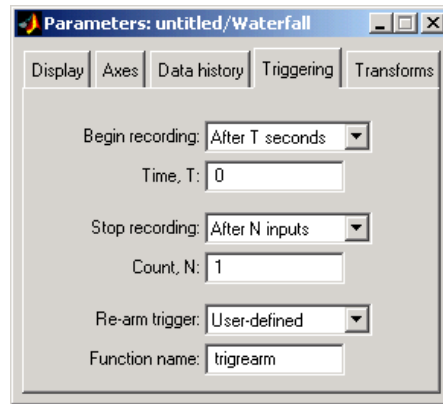
Begin recording

This parameter controls when the Waterfall block starts capturing data:

- Immediately — The Waterfall window captures the input data as soon as the simulation starts.

- After T seconds — The **Time, T** parameter appears in the dialog box. Enter the number of seconds the block should wait before it begins capturing data.

- After N inputs — The **Count, N** parameter appears in the dialog box. Enter the number of inputs the block should receive before it begins capturing data.

- User-defined — The **Function name** parameter appears in the dialog box. Enter the name of a MATLAB function that defines when the block should begin capturing data. For more information about how you define this function, see "Scope Trigger Function" on page 10-1218.

Stop recording

This parameter controls when the Waterfall block stops capturing data:

- `Never` — The block captures the input data as long as the simulation is running.

- `After T seconds` — The **Time, T** parameter appears in the dialog box. Enter the number of seconds the block should wait before it stops capturing data.

- `After N inputs` —The **Count, N** parameter appears in the dialog box. Enter the number of inputs the block should receive before it stops capturing data.

- `User-defined` — The **Function name** parameter appears in the dialog box. Enter the name of a MATLAB function that defines when the block should stop capturing data. For more information about how you define this function, see "Scope Trigger Function" on page 10-1218.

Re-arm trigger

This parameter controls when the Waterfall block begins waiting to capture data. It is available only when you select `After T seconds`, `After N inputs`, or `User-defined` for the **Stop recording** parameter:

- `Never` — The Waterfall Scope block starts and stops capturing data as defined by the **Begin recording** and **Stop recording** parameters.

- `After T seconds` — The **Time, T** parameter appears in the dialog box. Enter the number of seconds the block should wait before it begins waiting to capture data.

- `After N inputs` — The **Count, N** parameter appears in the dialog box. Enter the number of inputs the block should receive before it begins waiting to capture data.

- `User-defined` — The **Function name** parameter appears in the dialog box. Enter the name of a MATLAB function that defines when the block should begin waiting to capture data. For more information about how you define this function, see "Scope Trigger Function" on page 10-1218.

Waterfall

The triggering process is illustrated in the state diagram below.
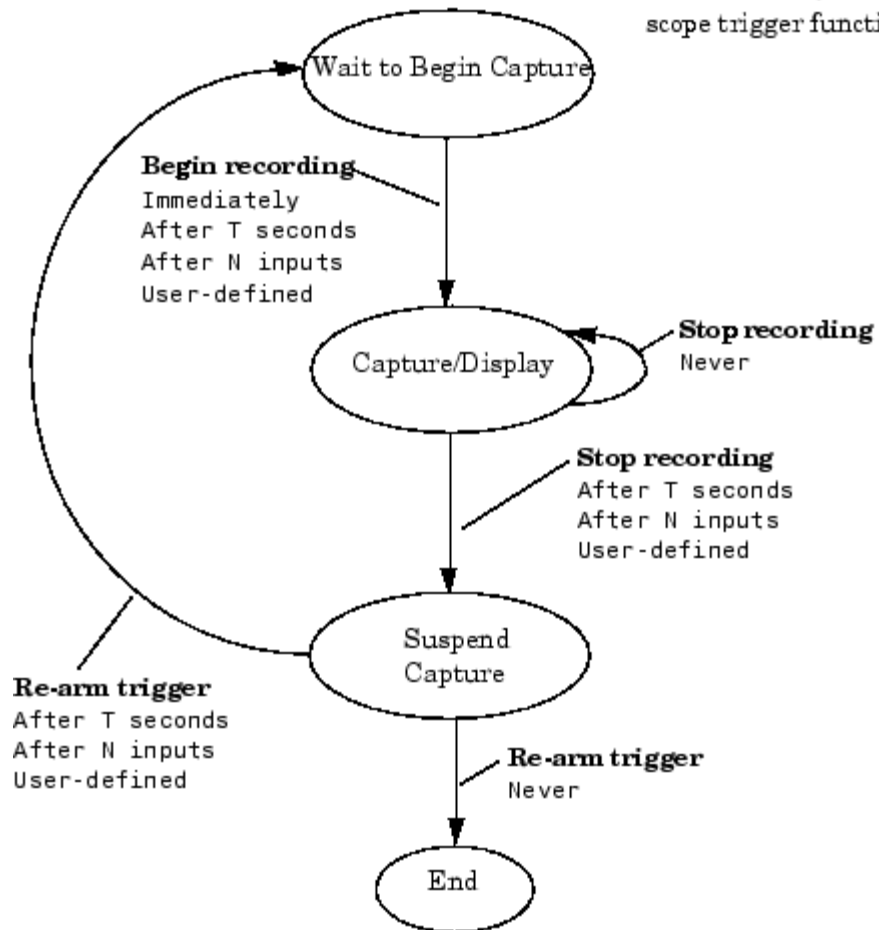
Scope Trigger Function

You can create custom scope trigger functions to control when the scope starts, stops, or begins waiting to capture data.
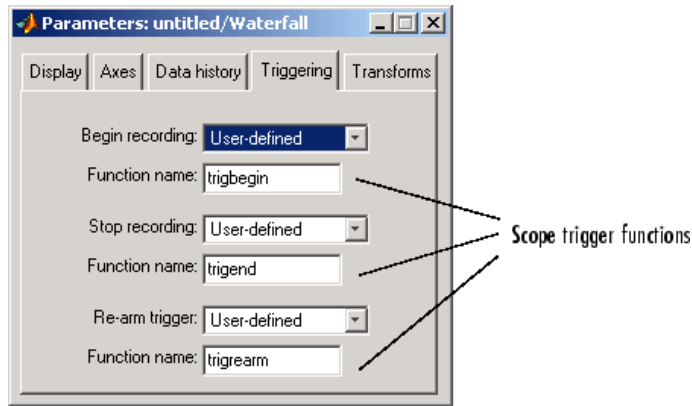
These functions must be valid MATLAB functions and be located either in the current directory or on the MATLAB path.

Each scope trigger function must have the following form

```
y = functionname(blk,t,u),
```

where `functionname` refers to the name you give your scope trigger function. The variable `blk` is the Simulink block handle. When the scope trigger function is called by the block, Simulink automatically populates this variable with the handle of the Waterfall block. The variable `t` is the current simulation time, represented by a real, double-precision, scalar value. The variable `u` is the vector input to the block. The output of the scope trigger function, `y`, is interpreted as a logical signal. It is either true or false:

- Begin recording scope trigger function

    - When the output of this scope trigger function is true, the Waterfall block starts capturing data.

    - When the output is false, the block remains in its current state.

- Stop recording scope trigger function

    - When the output of this scope trigger function is true, the block stops capturing data.

- When the output is false, the block remains in its current state.

- Re-arm trigger scope trigger function

  - When the output of this scope trigger function is true, the block waits for a begin recording event.

  - When the output is false, the block remains in its current state.

> **Note** The Waterfall block passes its input data directly to the scope trigger functions. These functions do not use the transformed data defined by the Transform parameters.

The following is an example of a scope trigger function. This function, called `trigPower` detects when the energy in u exceeds a certain threshold.

```
function y = trigPower(blk, t, u)

y = (u'*u > 2300);
```

The following is another example of a scope trigger function. This function, called `count3`, triggers the scope once three vectors with positive means are input to the block. Then, the function resets itself and begins searching for the next three input vectors with positive means. This scope trigger function is valid only when one Waterfall block is present in your model.

```
function y = count3(blk, t, u)

persistent state;
if isempty(state); state = 0; end
if mean(u)>0; state = state+1; end
y = (state>=3);
if y; state = 0; end
```

Transform Parameters

The following parameters transform the input data to the Waterfall block. The result of the transform is displayed in the Waterfall window.
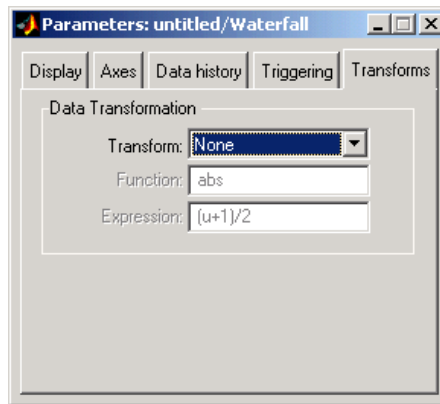
Note The block assumes that the input to the block corresponds to the **Transform** parameter you select. For example, when you choose Complex-> Angle, the block assumes that the input is complex. The block does not produce an error when the input is not complex. Therefore, you must verify the format of your input data to guarantee that a meaningful result is displayed in the Waterfall window.

Transform

Choose a transform that you would like to apply to the input of the Waterfall block:

- None — The input is displayed as it is received by the block.

- Amplitude-> dB — The block converts the input amplitude into decibels.

- Complex-> Mag Lin — The block converts the complex input into linear magnitude.

- `Complex-> Mag dB` — The block converts the complex input into magnitude in decibels.

- `Complex-> Angle` — The block converts the complex input into phase.

- `FFT-> Mag Lin Fs/2` — The block takes the linear magnitude of the FFT input and plots it from 0 to the Nyquist frequency.

- `FFT-> Mag dB Fs/2` — The block takes the magnitude of the FFT input, converts it to decibels, and plots it from 0 to the Nyquist frequency.

- `FFT-> Angle Fs/2` — The block converts the FFT input into phase and plots it from 0 to the Nyquist frequency.

- `Power-> dB` — The block converts the input power into decibels.

Function

This parameter is only available when you select `User-defined fcn` for the **Transform** parameter. Enter a function that you would like to apply to the input of the Waterfall block. For more information about how you define this function, see "Scope Transform Function" on page 10-1223.

Expression

This parameter is only available when you select `User-defined expr` for the **Transform** parameter. Enter an expression that you would like to apply to the input of the Waterfall block. The result of this expression must be real-valued. When you write the expression, be sure to include only one unknown variable. The block assumes this unknown variable represents the input to the block. When the block believes your expression is invalid, the following window appears.
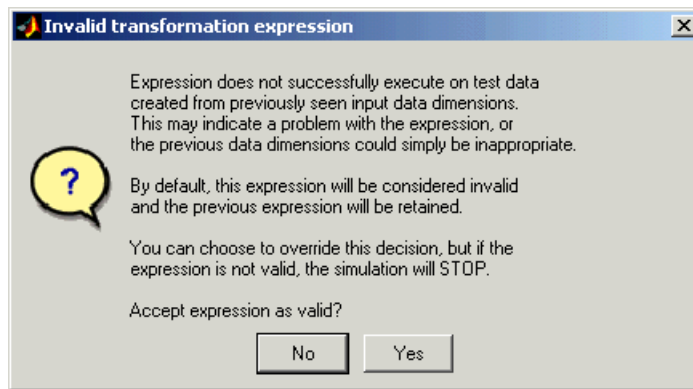
When you click **No**, your expression is not applied to the input. When you click **Yes** and your expression is invalid, your simulation stops and Simulink displays an error.

Scope Transform Function

You can create a scope transform function to control how the Waterfall block transforms your input data. This function must have a valid MATLAB function name and be located either in the current directory or on the MATLAB path.

Your scope transform function must have the following form

```
y = functionname(u),
```

where functionname refers to the name you give your function. The variable u is the real or complex vector input to the block. The output of the scope transform function, y, must be a double-precision, real-valued vector. When it is not, the simulation stops and Simulink displays an error. Note that the output vector does not need to be the same size as the input vector.

Examples   The following examples illustrate some capabilities of the Waterfall block.

Waterfall

Exporting Data

You can use the Waterfall block to export data to the MATLAB workspace or to SPTool:

1 Open and run the `dspanc` demo.

2 While the simulation is running, click the **Export to Workspace** button.

3 At the MATLAB command line, type `whos`.

The variable `ExportData` appears in your MATLAB workspace. `ExportData` is a 40-by-6 matrix. This matrix represents the six data vectors that were present in the Waterfall window at the time you clicked the **Export to Workspace** button. Each column of this matrix contains 40 filter coefficients. The columns of data were captured at six consecutive instants in time.

You can control what data is exported using the **Data logging** parameter in **Data history** pane of the Parameters dialog box. For more information, see "Data History Parameters" on page 10-1214.

4 While the simulation is running, click the **Export to SPTool** button.

The SPTool GUI opens and the variable `ExportData` is displayed in the **Signals** list.

For more information about SPTool, see the Signal Processing Toolbox documentation.

Capturing Data

You can use the Waterfall block to interact with your data while it is being captured:

1 Open and run the `dspanc` demo.

2 While the simulation is running, click the **Suspend data capture** button.

The Waterfall block no longer captures or displays the data coming from the Downsample block.

3 To continue capturing data, click the **Resume data capture** button.

4 To freeze the data display while continuing to capture data, click the **Snapshot display** button.

5 To view the Waterfall block that the data is coming from, click the **Go to scope block** button.

In the Simulink model window, the Waterfall block that corresponds to the active Waterfall window flashes. This feature is helpful when you have more than one Waterfall block in a model and you want to clarify which data is being displayed.

Linking Scopes

You can link several Waterfall blocks together in order to capture the effect of a model event in all of the Waterfall windows in the model:

1 Open the `dspanc` demo.

2 Drag a second Waterfall block into the demo model.

3 Connect this block to the Output port of the LMS Filter block as shown in the figure below.
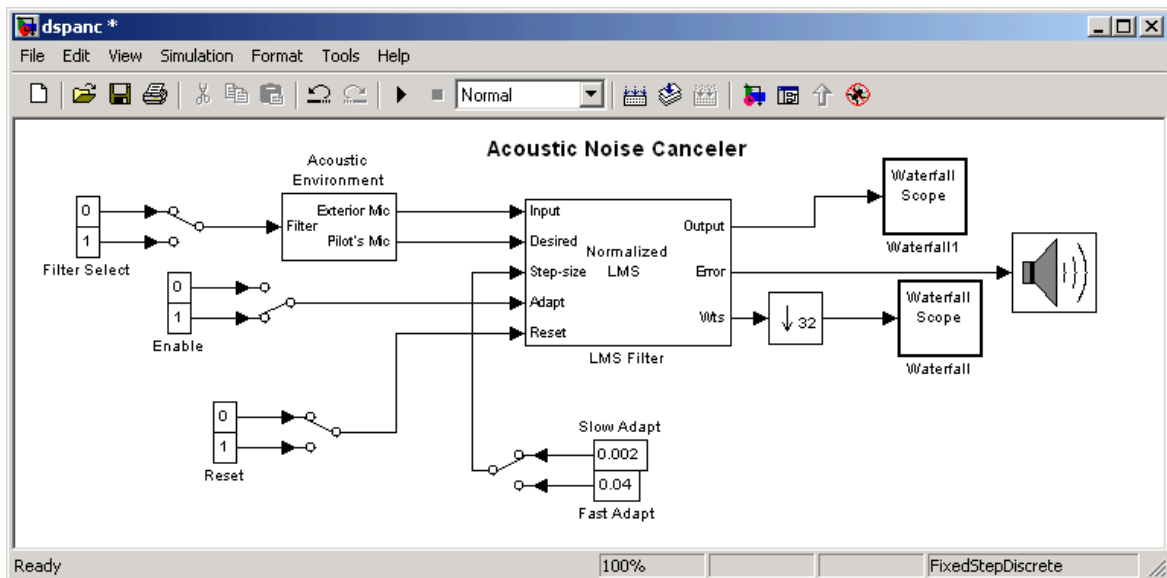
4 Run the model and view the model behavior in both Waterfall windows.

5 In the dspanc/Waterfall window, click the **Link scopes** button.

6 In the same window, click the **Suspend data capture** button.

The data capture is suspended in both scope windows.

7 Click the **Resume data capture** button.

The data capture resumes in both scope windows.

8 In the dspanc/Waterfall window, click the **Snapshot display** button.

In both scope windows, the data display freezes while the block continues to capture data.

9 To continue displaying the captured data, click the **Resume display** button.

Selecting Data

The following figure shows the Waterfall window displaying the output of the dspanc demo:
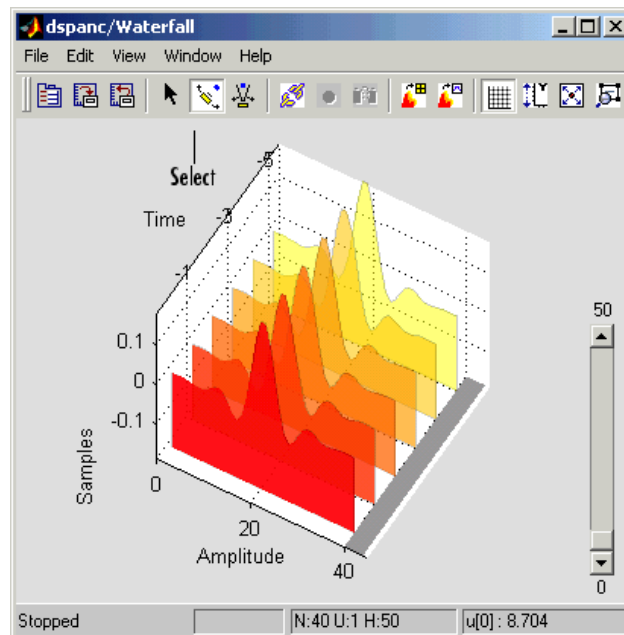
1 To select a particular set of data, click the **Select** button.

2 Click on the Time Bar at the bottom right of the axes to select a vector of data.

The Waterfall block highlights the selected trace.
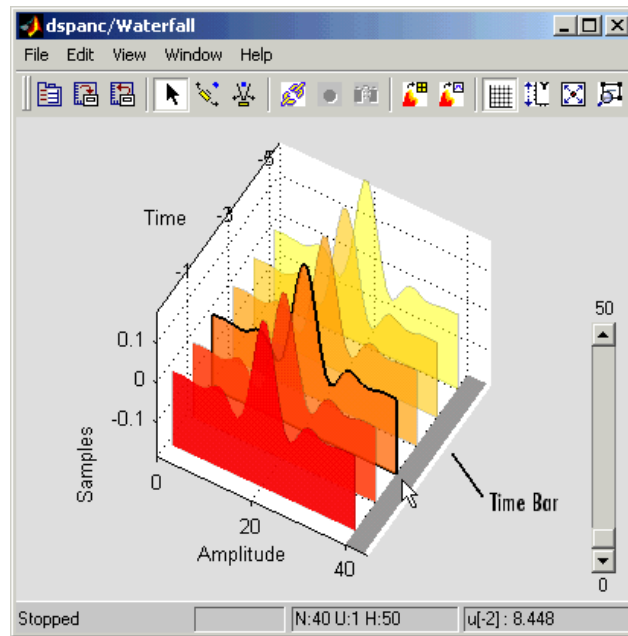
Relative index   Simulation time

While the simulation is running, in the bottom right corner, the Waterfall window displays the relative index of the selected trace. For example, in the previous figure, the selected vector is two sample times away from the most current data vector. When the simulation is stopped, the Waterfall window displays both the relative index and the simulation time associated with the selected trace.

3 To deselect the data vector, click it again.

4 Click-and-drag along the Time Bar.

Your selection follows the movement of the pointer.

You can use this feature to choose a particular vector to export to the MATLAB workspace or SPTool. For more information, see "Data History Parameters" on page 10-1214.

Zooming

You can use the Waterfall window to zoom in on data:

1 Click the **Zoom camera** button.

2 In the Waterfall window, click and hold down the left mouse button.

3 Move the mouse up and down and side-to-side to move closer and farther away from the axes.

4 To resize the axes to fit the Waterfall window, click the **Fit to view** button.

Rotating the Display

You can rotate the data displayed in the Waterfall window:

1 Click on the **Orbit camera** button.

2 In the Waterfall window, click and hold down the left mouse button.

3 Move the mouse in a circular motion to rotate the axes.

4 To return to the position of the original axes, click the **Restore scope position and view** button.

Scaling the Axes

You can use the Waterfall window to rescale the *y*-axis values:

1 Open and run the dspanc demo.

2 Click the **Rescale amplitude** button.

The *y*-axis changes so that its minimum value is zero. The maximum value is scaled to fit the data displayed.

Alternatively, you can scale the *y*-axis using the **Y Min** and **Y Max** parameters in the **Axes** pane of the Parameters dialog box. This is helpful when you want to undo the effects of rescaling the amplitude. For more information, see "Axes Parameters" on page 10-1213.

Saving Scope Settings

The Waterfall block can save the screen position and viewpoint of the Waterfall window:

1 Click the **Save scope position and view** button.

2 Close the Waterfall window.

3 Reopen the Waterfall window.

It reopens at the same place on your screen. The viewpoint of the axes also remains the same.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point
	• Single-precision floating point
	• Fixed point (signed and unsigned)
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers

The Waterfall block accepts any of these data types as input. However, the input is converted to double-precision before the block processes the data. The Waterfall block displays only real-valued, double-precision vectors of data. To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Scope	Simulink
Time Scope	Signal Processing Blockset
Vector Scope	Signal Processing Blockset
Spectrum Scope	Signal Processing Blockset
Matrix Viewer	Signal Processing Blockset
Signal To Workspace	Signal Processing Blockset
Triggered To Workspace	Signal Processing Blockset

Wavelet Analysis

Purpose   Decompose a signal into components of logarithmically decreasing frequency intervals and sample rates (requires the Wavelet Toolbox).

Library   dspobslib

Description
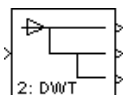
Note The Wavelet Analysis block is still supported but is likely to be obsoleted in a future release. We recommend replacing this block with the DWT block.

The Wavelet Analysis block uses the `wfilters` function from the Wavelet Toolbox to construct a dyadic analysis filter bank that decomposes a broadband signal into a collection of successively more bandlimited components. An n-level filter bank structure is shown below, where n is specified by the **Number of levels** parameter.

Wavelet Analysis Filter Bank, n Levels

$T_{s1} = T_s$
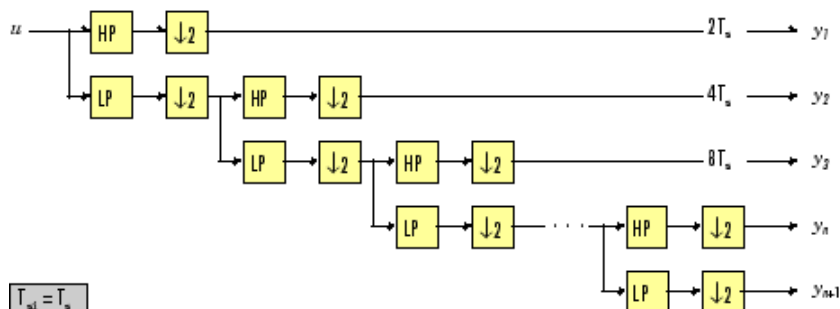
HP: highpass filter with $f_c \approx 1/2$ Nyquist
LP: lowpass filter with $f_c \approx 1/2$ Nyquist
$\downarrow 2$: downsample by 2

$T_{so} = (2^k)T_s$ for output y_k, $1 \leq k \leq n$
$T_{so} = (2^n)T_s$ for output y_{n+1}

At each level, the *low-frequency* output of the previous level is decomposed into adjacent high- and low-frequency subbands by a highpass (HP) and lowpass (LP) filter pair. Each of the two output subbands is half the bandwidth of the input to that level. The

bandlimited output of each filter is maximally decimated by a factor of 2 to preserve the bit rate of the original signal.

Filter Coefficients

The filter coefficients for the highpass and lowpass filters are computed by the Wavelet Toolbox function `wfilters`, based on the wavelet specified in the **Wavelet name** parameter. The table below lists the available options.

Wavelet Name	Sample Wavelet Function Syntax
Haar	`wfilters('haar')`
Daubechies	`wfilters('db4')`
Symlets	`wfilters('sym3')`
Coiflets	`wfilters('coif1')`
Biorthogonal	`wfilters('bior3.1')`
Reverse Biorthogonal	`wfilters('rbio3.1')`
Discrete Meyer	`wfilters('dmey')`

The **Daubechies**, **Symlets**, and **Coiflets** options enable a secondary **Wavelet order** parameter that allows you to specify the wavelet order. For example, if you specify a **Daubechies** wavelet with **Wavelet order** equal to 6, the Wavelet Analysis block calls the `wfilters` function with input argument `'db6'`.

The **Biorthogonal** and **Reverse Biorthogonal** options enable a secondary **Filter order [synthesis / analysis]** parameter that allows you to independently specify the wavelet order for the analysis and synthesis filter stages. For example, if you specify a **Biorthogonal** wavelet with **Filter order [synthesis / analysis]** equal to [2 / 6], the Wavelet Analysis block calls the `wfilters` function with input argument `'bior2.6'`.

Wavelet Analysis

See the Wavelet Toolbox documentation for more information about the `wfilters` function. If you want to explicitly specify the FIR coefficients for the analysis filter bank, use the Dyadic Analysis Filter Bank block.

Tree Structure

The wavelet tree structure has $n+1$ outputs, where n is the number of levels. The sample rate and bandwidth of the top output are half the input sample rate and bandwidth. The sample rate and bandwidth of each additional output (except the last) are half that of the output from the previous level. In general, for an input with sample period $T_{si} = T_s$, and bandwidth BW, output y_k has sample period $T_{so,k}$ and bandwidth BW_k.

$$T_{so,k} = \begin{cases} (2^k)T_s & (1 \leq k \leq n) \\ (2^n)T_s & (k = n+1) \end{cases}$$

$$BW_k = \begin{cases} \dfrac{BW}{2^k} & (1 \leq k \leq n) \\ \dfrac{BW}{2^n} & (k = n+1) \end{cases}$$

Note that in frame-based mode, the change in the sample period of output y_k is reflected by its frame size, $M_{o,k}$, rather than by its frame rate.

$$M_{o,k} = \begin{cases} \dfrac{M_i}{2^k} & (1 \leq k \leq n) \\ \dfrac{M_i}{2^n} & (k = n+1) \end{cases}$$

The bottom two outputs (y_n and y_{n+1}) share the same sample period, bandwidth, and frame size because they originate at the same tree level.

Sample-Based Operation

An *M*-by-*N* sample-based matrix input is treated as *M*N* independent channels, and the block filters each channel independently over time. The output at each port is the same size as the input, one output channel for each input channel. As described earlier, each output port has a different sample period.

The figure below shows the input and output sample periods for a 64-channel sample-based input to a three-level filter bank. The input has a period of 1, so the fastest output has a period of 2.
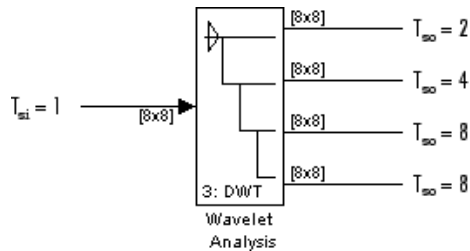
Frame-Based Operation

An M_i-by-*N* frame-based matrix input is treated as *N* independent channels, and the block filters each channel independently over time. The input frame size M_i must be a multiple of 2^n, and *n* is the number of filter bank levels. For example, a frame size of 8 would be appropriate for a three-level tree ($2^3=8$). The number of columns in each output is the same as the number of columns in the input.

Each output port has the *same frame period* as the input. The reduction in the output sample rates results from the smaller output frame sizes, as shown in the example below for a four-channel input to a three-level filter bank.

Wavelet Analysis
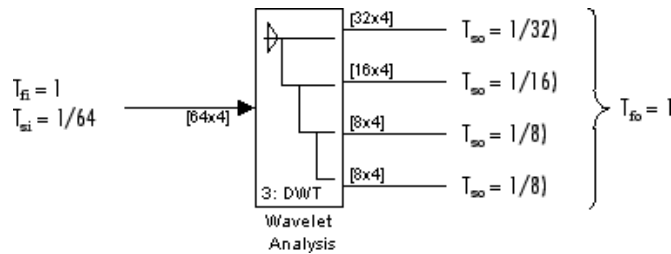
Zero Latency

The Wavelet Analysis block has *no tasking latency* for frame-based operation, which is always single-rate. The block therefore analyzes the first input sample (received at t=0) to produce the first output sample at each port.

Nonzero Latency

For sample-based operation, the Wavelet Analysis block is multirate and has 2^{n-1} samples of latency in both Simulink tasking modes. As a result, the block repeats a zero initial condition in each channel for the first 2^{n-1} output samples, before propagating the first analyzed input sample (computed from the input received at t=0).

Note For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" on page 2-56 and the topic on models with multiple sample rates in the Real-Time Workshop documentation.

Dialog Box
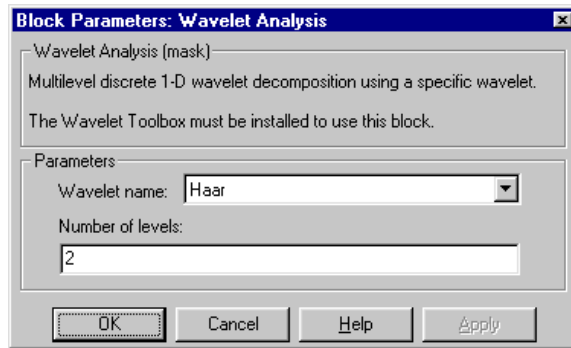
The parameters displayed in the dialog box vary for different wavelet types. Only some of the parameters listed below are visible in the dialog box at any one time.

Wavelet name
The wavelet used in the analysis.

Wavelet order
The order for the **Daubechies**, **Symlets**, and **Coiflets** wavelets. This parameter is available only when one of these wavelets is selected in the **Wavelet name** menu.

Filter order [synthesis / analysis]
The filter orders for the synthesis and analysis stages of the **Biorthogonal** and **Reverse Biorthogonal** wavelets. For example, [2 / 6] selects a second-order synthesis stage and a sixth-order analysis stage. The **Filter order** parameter is available only when one of the above wavelets is selected in the **Wavelet name** menu.

Number of levels
The number of filter bank levels. An n-level structure has $n+1$ outputs.

References Fliege, N. J. *Multirate Digital Signal Processing: Multirate Systems, Filter Banks, Wavelets*. West Sussex, England: John Wiley & Sons, 1994.

Wavelet Analysis

Strang, G. and T. Nguyen. *Wavelets and Filter Banks*. Wellesley, MA: Wellesley-Cambridge Press, 1996.

Vaidyanathan, P. P. *Multirate Systems and Filter Banks*. Englewood Cliffs, NJ: Prentice Hall, 1993.

Supported Data Types

- Double-precision floating point

See Also

Dyadic Analysis Filter Bank	DSP Blockset
Wavelet Synthesis	DSP Blockset
wfilters	Wavelet Toolbox

Purpose   Reconstruct a signal from its multirate bandlimited components
                   (requires the Wavelet Toolbox).

Library   dspobslib

Description
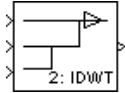
> **Note** The Wavelet Synthesis block is still supported but is likely to be
> obsoleted in a future release. We recommend replacing this block with
> the IDWT block.

The Wavelet Synthesis block uses the wfilters function from the
Wavelet Toolbox to reconstruct a signal that was decomposed by the
Wavelet Analysis block. The reconstruction or *synthesis* process is the
inverse of the analysis process, and restores the original signal by
upsampling, filtering, and summing the bandlimited inputs in stages
corresponding to the analysis process. An *n*-level synthesis filter bank
structure is shown below, where *n* is specified by the **Number of levels**
parameter.

Wavelet Synthesis Filter Bank, n Levels
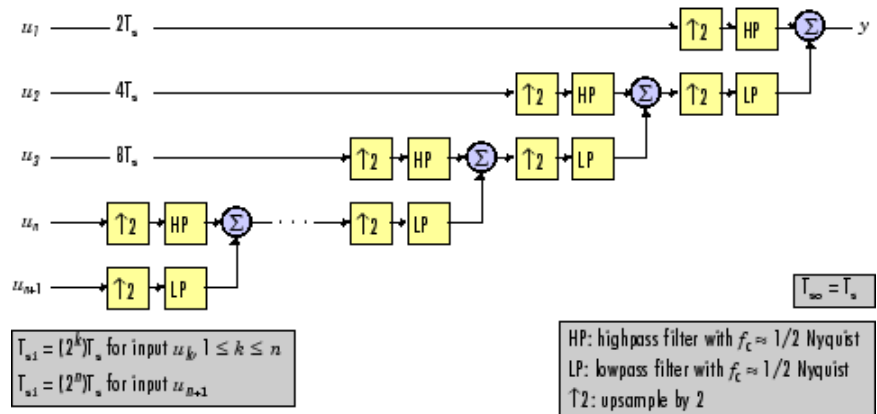
At each level, the two bandlimited inputs (one low-frequency, one
high-frequency, both with the same sample rate) are upsampled by

a factor of 2 to match the sample rate of the input to the next stage. They are then filtered by a highpass (HP) and lowpass (LP) filter pair with coefficients calculated to cancel (in the subsequent summation) the aliasing introduced in the corresponding analysis filter stage. The output from each (upsample-filter-sum) level has twice the bandwidth and twice the sample rate of the input to that level.

For perfect reconstruction, the Wavelet Synthesis and Wavelet Analysis blocks must have the same parameter settings.

Filter Coefficients

The filter coefficients for the highpass and lowpass filters are computed by the Wavelet Toolbox function `wfilters`, based on the wavelet specified in the **Wavelet name** parameter. The table below lists the available options.

Wavelet Name	Sample Wavelet Function Syntax
Haar	`wfilters('haar')`
Daubechies	`wfilters('db4')`
Symlets	`wfilters('sym3')`
Coiflets	`wfilters('coif1')`
Biorthogonal	`wfilters('bior3.1')`
Reverse Biorthogonal	`wfilters('rbio3.1')`
Discrete Meyer	`wfilters('dmey')`

The **Daubechies**, **Symlets**, and **Coiflets** options enable a secondary **Wavelet order** parameter that allows you to specify the wavelet order. For example, if you specify a **Daubechies** wavelet with **Wavelet order** equal to 6, the Wavelet Synthesis block calls the `wfilters` function with input argument `'db6'`.

The **Biorthogonal** and **Reverse Biorthogonal** options enable a secondary **Filter order [synthesis / analysis]** parameter that allows

you to independently specify the wavelet order for the analysis and synthesis filter stages. For example, if you specify a **Biorthogonal** wavelet with **Filter order [synthesis / analysis]** equal to [2 / 6], the Wavelet Synthesis block calls the `wfilters` function with input argument `'bior2.6'`.

See the Wavelet Toolbox documentation for more information about the `wfilters` function. If you want to explicitly specify the FIR coefficients for the synthesis filter bank, use the Dyadic Synthesis Filter Bank block.

Tree Structure

The wavelet tree structure has $n+1$ inputs, where n is the number of levels. The sample rate and bandwidth of the output are twice the sample rate and bandwidth of the top input. The sample rate and bandwidth of each additional input (except the last) are half that of the input to the previous level.

$$T_{si,\,k+1} = 2\,T_{si,\,k} \qquad 1 \le k < n$$

$$BW_{k+1} = \frac{BW_k}{2} \qquad 1 \le k < n$$

The bottom two inputs (u_n and u_{n+1}) should have the same sample rate and bandwidth since they are processed by the same level.

$$T_{si,\,n+1} = T_{si,\,n}$$

$$BW_{n+1} = BW_n$$

Note that in frame-based mode, the sample period of input u_k is reflected by its frame size, $M_{i,k}$, rather than by its frame rate.

$$M_{i,\,k+1} = \frac{M_{i,\,k}}{2} \qquad 1 \le k < n$$

$$M_{i,\,n+1} = M_{i,\,n}$$

Wavelet Synthesis

Sample-Based Operation

An *M*-by-*N* sample-based matrix input is treated as $M*N$ independent channels, and the block filters each channel independently over time. The output is the same size as the input at each port, one output channel for each input channel. As described earlier, each input port has a different sample period.

The figure below shows the input and output sample periods for the four 64-channel sample-based inputs to a three-level filter bank. The fastest input has a period of 2, so the output period is 1.
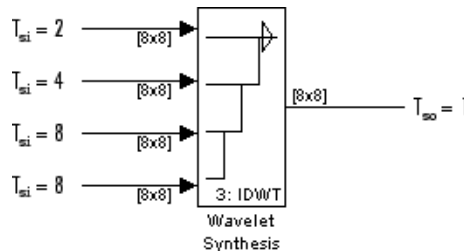
Frame-Based Operation

An M_i-by-*N* frame-based matrix input is treated as *N* independent channels, and the block filters each channel independently over time. The number of columns in the output is the same as the number of columns in the input.

All inputs must have the same frame period, which is also the output frame period. The different input sample rates should be represented by the input frame sizes: If the input to the top port has frame size M_i, the input to the second-from-top port should have frame size $M_i/2$, the input to the third-from-top port should have frame size $M_i/4$, and so on. The input to the *bottom* port should have the same frame size as the second-from-bottom port. The increase in the sample rate of the output is also represented by its frame size, which is twice the largest input frame size.

The relationship between sample periods, frame periods, and frame sizes is shown below for a four-channel frame-based input to a 3-level filter bank.
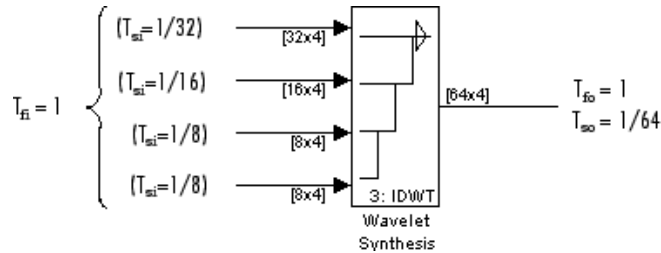
Zero Latency

The Wavelet Synthesis block has *no tasking latency* for frame-based operation, which is always single-rate. The block therefore uses the first input samples (received at t=0) to synthesize the first output sample.

Nonzero Latency

For sample-based operation, the Wavelet Synthesis block is multirate and has the following tasking latencies:

- 2^n-2 samples in Simulink's single-tasking mode

- 2^n samples in Simulink's multitasking mode

In the above cases, the block repeats a zero initial condition in each channel for the first D output samples, where D is the latency shown above. For example, in single-tasking mode the block generates 2^n-2 zero-valued output samples in each channel before propagating the first synthesized output sample (computed from the inputs received at t=0).

Note For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" on page 2-56 and the topic on models with multiple sample rates in the Real-Time Workshop documentation.

Wavelet Synthesis

Dialog Box
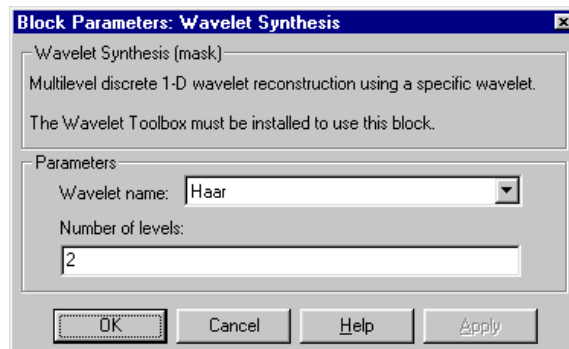
The parameters displayed in the dialog box vary for different wavelet types. Only some of the parameters listed below are visible in the dialog box at any one time.

Wavelet name

> The wavelet used in the synthesis.

Wavelet order

> The order for the **Daubechies**, **Symlets**, and **Coiflets** wavelets. This parameter is available only when one of these wavelets is selected in the **Wavelet name** menu.

Filter order [synthesis / analysis]

> The filter orders for the synthesis and analysis stages of the **Biorthogonal** and **Reverse Biorthogonal** wavelets. For example, [2 / 6] selects a second-order synthesis stage and a sixth-order analysis stage. The **Filter order** parameter is available only when one of the above wavelets is selected in the **Wavelet name** menu.

Number of levels

> The number of filter bank levels. An *n*-level structure has *n*+1 outputs.

References   Fliege, N. J. *Multirate Digital Signal Processing: Multirate Systems, Filter Banks, Wavelets*. West Sussex, England: John Wiley & Sons, 1994.

Strang, G. and T. Nguyen. *Wavelets and Filter Banks*. Wellesley, MA: Wellesley-Cambridge Press, 1996.

Vaidyanathan, P. P. *Multirate Systems and Filter Banks*. Englewood Cliffs, NJ: Prentice Hall, 1993.

Supported Data Types

- Double-precision floating point

See Also

Dyadic Synthesis Filter Bank	DSP Blockset
Wavelet Analysis	DSP Blockset
wfilters	Wavelet Toolbox

Window Function

Purpose   Compute and/or apply window to input signal

Library   Signal Operations
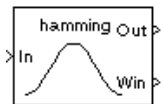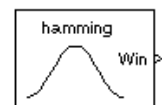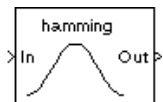
Description   The Window Function block computes a window, and/or applies a window to an input signal. This block supports real and complex floating-point and fixed-point inputs.

Operation Modes

The Window Function block has three modes of operation that you can select via the **Operation** parameter. In each mode, the block first creates a window vector w by sampling the window specified in the **Window type** parameter at M discrete points. The operation modes are

- Apply window to input

  In this mode, the block computes an M-by-1 window vector w and multiplies it element-wise with each of the N channels in the M-by-N input matrix u. This is equivalent to the following MATLAB code.

  ```
  y = repmat(w,1,N) .* u          % Equivalent MATLAB code
  ```

  In this mode, a length-M 1-D vector input is treated as an M-by-1 matrix. The output y always has the same dimension as the input. When the input is frame based, the output is frame based; otherwise, the output is sample based.

- Generate window

  In this mode the block generates a sample-based 1-D window vector w with length M specified by the **Window length** parameter. The In port is disabled for this mode.

- Generate and apply window

  In this mode, the block computes an M-by-1 window vector w and multiplies it element-wise with each of the N channels in the M-by-N input matrix u. This is equivalent to the following MATLAB code.

```
y = repmat(w,1,N) .* u          % Equivalent MATLAB code
```

In this mode, a length-*M* 1-D vector input is treated as an *M*-by-1 matrix. The block produces two outputs:

- At the Out port, the block produces the result of the multiplication y, which has the same dimension as the input. When the input is frame based, the output y is frame based; otherwise, the output y is sample based.

- At the Win port, the block produces the *M*-by-1 window vector w. Output w is always sample based.

Window Type

The available window types are shown in the table below. For complete information about the window functions, consult the Signal Processing Toolbox documentation.

Window Type	Description
Bartlett	Computes a Bartlett window. `w = bartlett(M)`
Blackman	Computes a Blackman window. `w = blackman(M)`
Boxcar	Computes a rectangular window. `w = rectwin(M)`
Chebyshev	Computes a Chebyshev window with stopband ripple R. `w = chebwin(M,R)`

Window Function

Window Type	Description
Hamming	Computes a Hamming window. `    w = hamming(M)`
Hann	Computes a Hann window (also known as a Hanning window). `    w = hann(M)`
Hanning	Obsolete. This window option is included only for compatibility with older models. Use the `Hann` option instead of `Hanning` whenever possible.
Kaiser	Computes a Kaiser window with Kaiser parameter `beta`. `    w = kaiser(M,beta)`
Triang	Computes a triangular window. `    w = triang(M)`

User Defined	Computes the user-defined window function specified by the entry in the **Window function name** parameter, `usrwin`.
	
	`    w = usrwin(M) % Window takes no extra parameters`
	`    w = usrwin(M,x`$_1$`,...,x`$_n$`) % Window takes extra`
	`    parameters {x`$_1$` ... x`$_n$`}`

Window Sampling

For the generalized-cosine windows (`Blackman`, `Hamming`, `Hann`, and `Hanning`), the **Sampling** parameter determines whether the window samples are computed in a *periodic* or a *symmetric* manner. For example, when **Sampling** is set to `Symmetric`, a Hamming window of length *M* is computed as

```
w = hamming(M)     % Symmetric (aperiodic) window
```

When **Sampling** is set to `Periodic`, the same window is computed as

```
w = hamming(M+1)     % Periodic (asymmetric) window
w = w(1:M)
```

Fixed-Point Data Types

The following diagram shows the data types used within the Window block for fixed-point signals for each of the three operating modes.
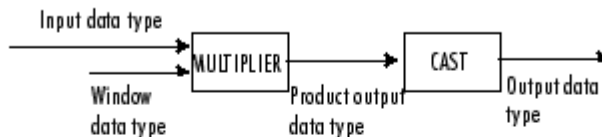
Apply window to input

The input data type comes from the driving block. You can set the window, product output, and output data types in the block dialog. In this mode, the window vector is not output from the block.

Generate window

In this mode, the block acts as a source. The window vector is output in the window data type you specify in the block dialog.

Generate and apply window
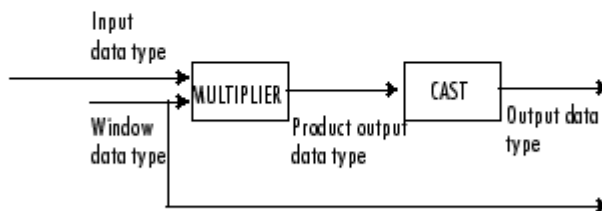
The input data type comes from the driving block. You can set the window, product output, and output data types in the block dialog. In this mode, the window vector is output from the block.

You can set the window, product output, and output data types in the block dialog as discussed below.

Dialog Box

The **Main** pane of the Window Function block dialog appears as follows:
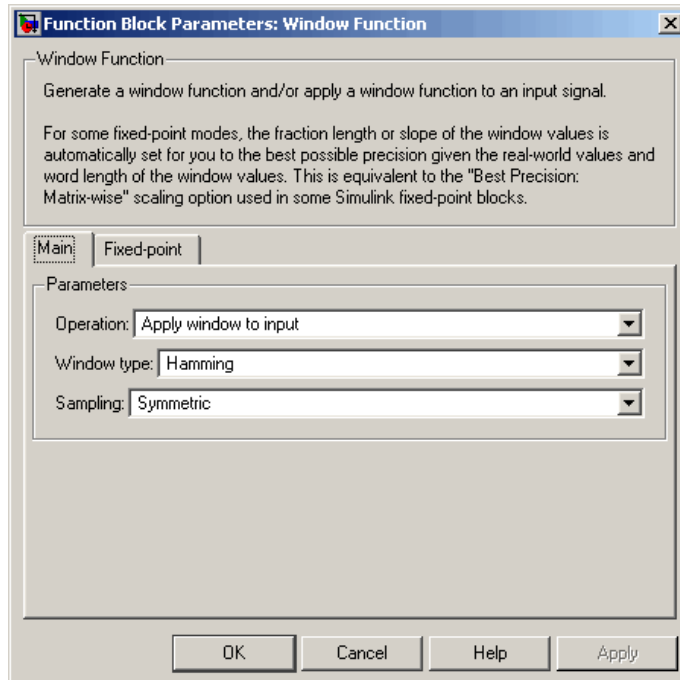
Operation

Specify the block's operation as discussed in "Operation Modes" on page 10-1246. The port configuration of the block is updated to match the setting of this parameter.

Window type

Specify the type of window to apply as listed in "Window Type" on page 10-1247. Tunable.

Sample Mode

Specify the sample mode for the block, Continuous or Discrete, when it is in Generate Window mode. In the Apply window to output and Generate and apply window modes, the block

Window Function

inherits the sample time from its driving block. Therefore, this parameter is only visible when you select `Generate window` for the **Operation** parameter.

Sample time

Specify the sample time for the block when it is in `Generate window` and `Discrete` modes. In `Apply window to output` and `Generate and apply window` modes, the block inherits the sample time from its driving block. This parameter is only visible when you select `Discrete` for the **Sample Mode** parameter.

Window length

Specify the length of the window to apply. This parameter is only visible when you select `Generate window` for the **Operation** parameter. Otherwise, the window vector length is computed to match the input frame size, M.

Sampling

Specify the window sampling for generalized-cosine windows. This parameter is only visible when you select `Blackman`, `Hamming`, `Hann`, or `Hanning` for the **Window type** parameter. Tunable.

Stopband attenuation in dB

Specify the level of stopband attenuation, R_s, in decibels. This parameter is only visible when you select `Chebyshev` for the **Window type** parameter. Tunable.

Beta

Specify the `Kaiser` window β parameter. Increasing β widens the mainlobe and decreases the amplitude of the window sidelobes in the window's frequency magnitude response. This parameter is only visible when you select `Kaiser` for the **Window type** parameter. Tunable.

Window function name

Specify the name of the user-defined window function to be calculated by the block. This parameter is only visible when you select `User defined` for the **Window type** parameter.

Specify additional arguments to the hamming function
Select to enable the **Cell array of additional arguments** parameter, when the user-defined window requires parameters other than the window length. This parameter is only visible when you select User defined for the **Window type** parameter.

Cell array of additional arguments
Specify the extra parameters required by the user-defined window function, besides the window length. This parameter is only available when you select the **Specify additional arguments to the hamming function** parameter. The entry must be a cell array.

The **Data types** pane of the Window block dialog is discussed in the following sections:

"Parameters for Generate Window Only Mode" on page 10-1253

"Parameters for Apply Window Modes" on page 10-1256

Parameters for Generate Window Only Mode

The following parameters are available when the **Operation** parameter is set to Generate window:

Window Function
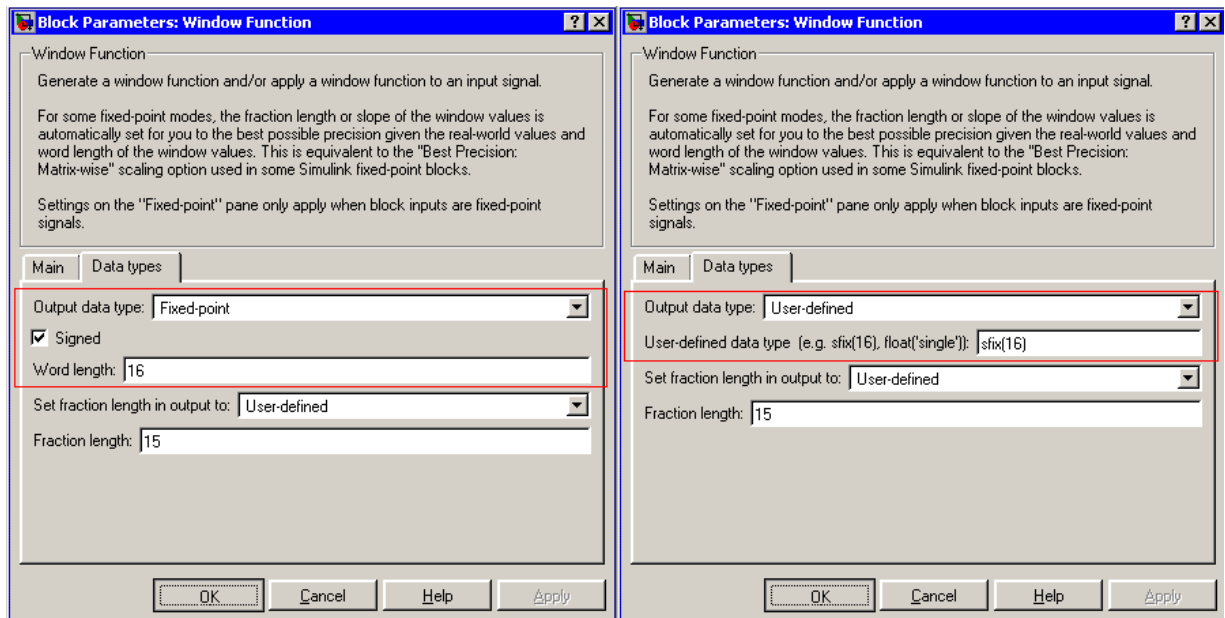
Output data type

Specify the output data type in one of the following ways:

- Choose `double` or `single` from the list

- Choose `Fixed-point` to specify the output data type and scaling in the **Signed**, **Word length**, **Set fraction length in output to,** and **Fraction length** parameters

- Choose `User-defined` to specify the output data type and scaling in the **User-defined data type**, **Set fraction length in output to,** and **Fraction length** parameters

- Choose `Inherit via back propagation` to set the output data type and scaling to match the following block

Signed

Select to output a signed fixed-point signal. Otherwise, the signal will be unsigned.

Word length

Specify the word length, in bits, of the fixed-point output data type. This parameter is only visible when you select `Fixed-point` for the **Output data type** parameter.

User-defined data type

Specify any built-in or fixed-point data type. You can specify fixed-point data types using the `sfix`, `ufix`, `sint`, `uint`, `sfrac`, and `ufrac` functions from Simulink Fixed Point. This parameter is only visible when you select `User-defined` for the **Output data type** parameter.

Set fraction length in output to

Specify the scaling of the fixed-point output by either of the following two methods:

- Choose `Best precision` to have the output scaling automatically set such that the output signal has the best possible precision.

- Choose `User-defined` to specify the output scaling in the **Fraction length** parameter.

This parameter is only visible when you select `Fixed-point` or `User-defined` for the **Output data type** parameter, and when the specified output data type is a fixed-point data type.

Fraction length

Specify the fraction length, in bits, of the fixed-point output data type. This parameter is only visible when you select `Fixed-point` or `User-defined` for the **Output data type** parameter and `User-defined` for the **Set fraction length in output to** parameter.

Window Function

Parameters for Apply Window Modes

The following parameters are available on the Fixed-Point pane when the **Operation** parameter is set to either `Apply window to input` or `Generate and apply window`.
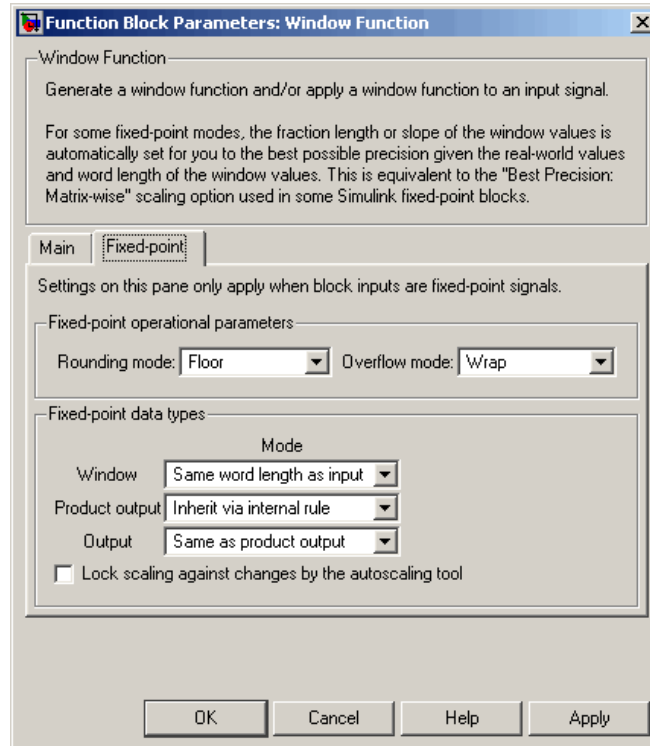
Rounding mode

Select the rounding mode for fixed-point operations.

The window vector w does not obey this parameter; it always rounds to `Nearest`.

Overflow mode

Select the overflow mode for fixed-point operations.

The window vector w does not obey this parameter; it is always saturated.

Window

Choose how you will specify the word length and fraction length of the window vector w.

When you select Same word length as input, the word length of the window vector elements is the same as the word length of the input. The fraction length is automatically set to the best precision possible.

When you select Specify word length, you are able to enter the word length of the window vector elements in bits. The fraction length is automatically set to the best precision possible.

When you select Binary point scaling, you are able to enter the word length and the fraction length of the window vector elements in bits.

When you select Slope and bias scaling, you are able to enter the word length, in bits, and the slope of the window vector elements. This block requires power-of-two slope and a bias of zero.

The window vector does not obey the **Rounding mode** and **Overflow mode** parameters; it is always saturated and rounded to Nearest.

Product output

Use this parameter to specify how you would like to designate the product output word and fraction lengths. Refer to "Fixed-Point Data Types" on page 10-1153 for illustrations depicting the use of the product output data type in this block:

- When you select `Inherit via internal rule`, the product output word length and fraction length are automatically set according to the following equations:

$$ideal\ product\ output\ word\ length =$$
$$input\ word\ length + window\ coefficients\ word\ length$$

$$ideal\ product\ output\ fraction\ length =$$
$$input\ fraction\ length + window\ coefficients\ fraction\ length$$

Note The actual product output word length may be equal to or greater than the calculated ideal product output word length, depending on the settings on the **Hardware Implementation** pane of the Configuration Parameters dialog box.

- When you select `Same as input`, these characteristics will match those of the input to the block.
- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the product output, in bits.
- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the product output. This block requires power-of-two slope and a bias of zero.

Output
Choose how you will specify the word length and fraction length of the output of the block:

- When you select `Same as product output`, these characteristics will match those of the product output.
- When you select `Same as input`, these characteristics will match those of the input to the block.

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the output, in bits.

- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the output. This block requires power-of-two slope and a bias of zero.

Lock scaling against changes by the autoscaling tool
Select this parameter to prevent any fixed-point scaling you specify in this block mask from being overriden by the autoscaling tool in the Fixed-Point Settings interface. For more information about the autoscaling tool, refer to "Fixed-Point Settings Interface" on page 8-28.

Supported Data Types

- Double-precision floating point

- Single-precision floating point

- Fixed point (signed only)

- 8-, 16-, and 32-bit signed integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

FFT	Signal Processing Blockset
bartlett	Signal Processing Toolbox
blackman	Signal Processing Toolbox
rectwin	Signal Processing Toolbox
chebwin	Signal Processing Toolbox
hamming	Signal Processing Toolbox
hann	Signal Processing Toolbox

Window Function

kaiser	Signal Processing Toolbox
triang	Signal Processing Toolbox

Purpose   Compute estimate of autoregressive (AR) model parameters using Yule-Walker method

Library   Estimation / Parametric Estimation
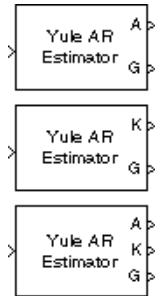
Description   The Yule-Walker AR Estimator block uses the Yule-Walker AR method, also called the autocorrelation method, to fit an autoregressive (AR) model to the windowed input data by minimizing the forward prediction error in the least squares sense. This formulation leads to the Yule-Walker equations, which are solved by the Levinson-Durbin recursion. Block outputs are always nonsingular.

The Yule-Walker AR Estimator block can output the AR model coefficients as polynomial coefficients, reflection coefficients, or both. The input is a sample-based vector (row, column, or 1-D) or frame-based vector (column only) representing a frame of consecutive time samples from a single-channel signal, which is assumed to be the output of an AR system driven by white noise. The block computes the normalized estimate of the AR system parameters, $A(z)$, independently for each successive input frame.

$$H(z) = \frac{\sqrt{G}}{A(z)} = \frac{\sqrt{G}}{1 + a(2)z^{-1} + \ldots + a(p+1)z^{-p}}$$

When you select **Inherit estimation order from input dimensions**, the order, p, of the all-pole model is one less than the length of the input vector. Otherwise, the order is the value specified by the **Estimation order** parameter. To guarantee a valid output, you must set the **Estimation order** parameter to be less than or equal to half the input vector length. The Yule-Walker AR Estimator and Burg AR Estimator blocks return similar results for large frame sizes.

When **Output(s)** is set to A, port A is enabled. Port A outputs a column vector of length $p+1$ that contains the normalized estimate of the AR model coefficients in descending powers of z

```
[1 a(2) ... a(p+1)]
```

When **Output(s)** is set to K, port K is enabled. Port K outputs a length-p column vector whose elements are the AR model reflection coefficients. When **Output(s)** is set to A and K, both port A and K are enabled, and each port outputs its respective column vector of AR model coefficients. The outputs at both ports A and K are always 1-D vectors.

The square of the model gain, *G (a scalar)*, is provided at port G.

See the Burg AR Estimator block reference page for a comparison of the Burg AR Estimator, Covariance AR Estimator, Modified Covariance AR Estimator, and Yule-Walker AR Estimator blocks.

Dialog Box
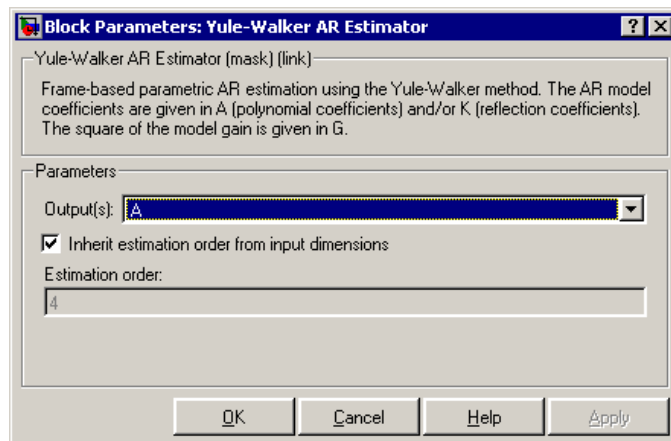
Output(s)
> The type of AR model coefficients output by the block. The block can output polynomial coefficients (A), reflection coefficients (K), or both (A and K). Nontunable.

Inherit estimation order from input dimensions
> When selected, sets the estimation order p to one less than the length of the input vector. Nontunable.

Estimation order

> The order of the AR model, *p*. This parameter is enabled when you do not select **Inherit estimation order from input dimensions**. Nontunable.

References   Kay, S. M. *Modern Spectral Estimation: Theory and Application.* Englewood Cliffs, NJ: Prentice-Hall, 1988.

Marple, S. L., Jr., *Digital Spectral Analysis with Applications.* Englewood Cliffs, NJ: Prentice-Hall, 1987.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
A	• Double-precision floating point • Single-precision floating point
K	• Double-precision floating point • Single-precision floating point
G	• Double-precision floating point • Single-precision floating point

The output data type is the same as the input data type. To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Burg AR Estimator	Signal Processing Blockset
Covariance AR Estimator	Signal Processing Blockset

Yule-Walker AR Estimator

Modified Covariance AR Estimator	Signal Processing Blockset
Yule-Walker Method	Signal Processing Blockset
`aryule`	Signal Processing Toolbox

Purpose   Design and apply an IIR filter.

Library   dspobslib

Description

yulewalk

> **Note** The Yule-Walker IIR Filter Design block is still supported but
> is likely to be obsoleted in a future release. We strongly recommend
> replacing this block with the Digital Filter block.

The Yule-Walker IIR Filter Design block designs a recursive (ARMA)
digital filter with arbitrary multiband magnitude response, and applies
it to a discrete-time input using the Direct-Form II Transpose Filter
block. The filter design, which uses the yulewalk function in the
Signal Processing Toolbox, performs a least-squares fit to the specified
frequency response.

An *M*-by-*N* sample-based matrix input is treated as *M**N independent
channels, and an *M*-by-*N* frame-based matrix input is treated as *N*
independent channels. In both cases, the block filters each channel
independently over time, and the output has the same size and frame
status as the input.

The **Band-edge frequency vector** parameter is a vector of frequency
points in the range 0 to 1, where 1 corresponds to half the sample
frequency. The first element of this vector must be 0 and the last
element 1, and intermediate points must appear in ascending order.
The **Magnitudes at these frequencies** parameter is a vector
containing the desired magnitude response at the points specified in the
Band-edge frequency vector.

Note that, unlike the Remez FIR Filter Design block, each
frequency-magnitude pair specifies the junction of two adjacent
frequency bands, so there are no "don't care" regions.

Yule-Walker IIR Filter Design

Band edge frequency = [0.0 0.3 0.4 0.6 0.7 1.0]

Magnitudes [0.0 1.0 0.0 0.0 0.5 0.5]
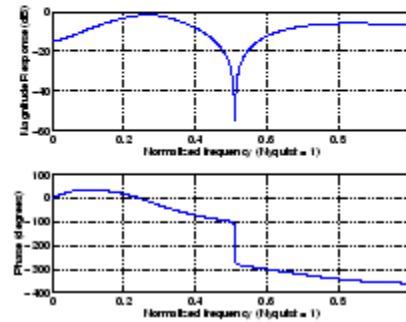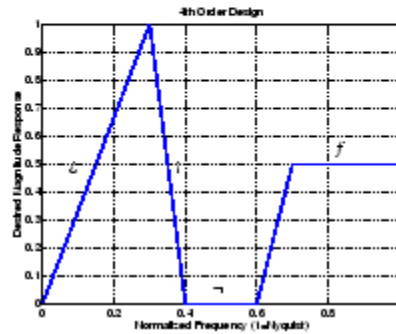
When specifying the **Band-edge frequency vector** and **Magnitudes at these frequencies** vectors, avoid excessively sharp transitions from passband to stopband. You may need to experiment with the slope of the transition region to get the best filter design.

For more details on the Yule-Walker filter design algorithm, see the description of the `yulewalk` function in the Signal Processing Toolbox documentation.

Dialog Box

Function Block Parameters: Yule-Walker IIR Filter Design

Yule-Walker IIR Filter Design (mask) (link)

Yule-Walker IIR filter design.

Parameters

Filter order:

8

Band-edge frequency vector (including 0 and 1):

[0 .4 .6 1]

Magnitudes at these frequencies:

[1 1 0 0]

| OK | Cancel | Help | Apply |
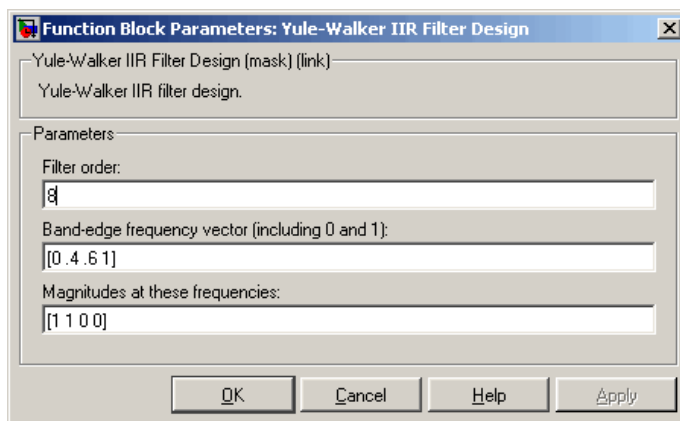
Filter order

The order of the filter.

Band-edge frequency vector

A vector of frequency points. The value 1 corresponds to half the sample frequency. The first element of this vector must be 0 and the last element 1. Tunable.

Magnitudes at these frequencies

A vector of frequency response magnitudes corresponding to the points in the **Band-edge frequency vector**. This vector must be the same length as the **Band-edge frequency vector**. Tunable.

References   Oppenheim, A. V. and R. W. Schafer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989.

Proakis, J. and D. Manolakis. *Digital Signal Processing*. 3rd ed. Englewood Cliffs, NJ: Prentice-Hall, 1996.

Yule-Walker Method

Purpose   Compute parametric estimate of the spectrum using Yule-Walker
              autoregressive (AR) method

Library   Estimation / Power Spectrum Estimation
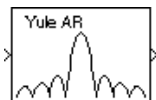
Description   The Yule-Walker Method block estimates the power spectral density
              (PSD) of the input using the Yule-Walker AR method. This method, also
              called the autocorrelation method, fits an autoregressive (AR) model to
              the windowed input data by minimizing the forward prediction error
              in the least squares sense. This formulation leads to the Yule-Walker
              equations, which are solved by Levinson-Durbin recursion. Block
              outputs are always nonsingular.

              The input is a sample-based vector (row, column, or 1-D) or frame-based
              vector (column only) representing a frame of consecutive time samples
              from a single-channel signal. The block's output (a column vector) is
              the estimate of the signal's power spectral density at N_{fft} equally spaced
              frequency points in the range $[0,F_s)$, where F_s is the signal's sample
              frequency.

              When you select **Inherit estimation order from input dimensions**,
              the order of the all-pole model is one less that the input frame size.
              Otherwise, the order is the value specified by the **Estimation order**
              parameter. To guarantee a valid output, you must set the **Estimation
              order** parameter to be less than or equal to half the input vector
              length. The spectrum is computed from the FFT of the estimated AR
              model parameters.

              When you select **Inherit FFT length from estimation order**, N_{fft} is
              specified by (estimation order + 1), which must be a power of 2. When
              you do *not* select **Inherit FFT length from estimation order**, N_{fft} is
              specified as a power of 2 by the **FFT length** parameter, and the block
              zero pads or truncates the input to N_{fft} before computing the FFT. The
              output is always sample based.

              See the Burg Method block reference for a comparison of the Burg
              Method, Covariance Method, Modified Covariance Method, and
              Yule-Walker AR Estimator blocks. The Yule-Walker AR Estimator and
              Burg Method blocks return similar results for large buffer lengths.

Dialog Box
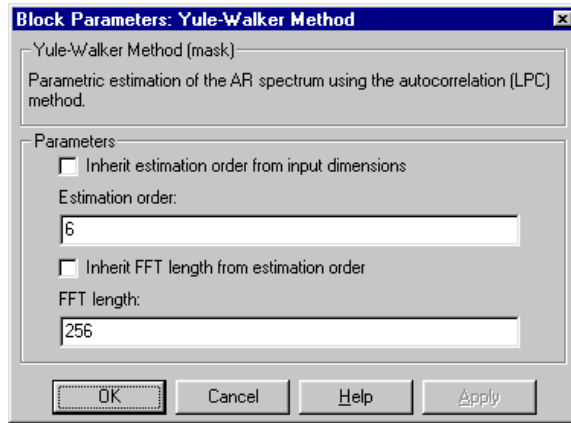
Inherit estimation order from input dimensions
When selected, sets the estimation order to one less than the length of the input vector.

Estimation order
The order of the AR model. This parameter is enabled when you do not select **Inherit estimation order from input dimensions**.

Inherit FFT length from estimation order
When selected, uses the estimation order to determine the number of data points, N_{fft}, on which to perform the FFT. Sets N_{fft} equal to (estimation order + 1). Note that N_{fft} must be a power of 2, so (estimation order + 1) must be a power of 2.

FFT length
The number of data points, N_{fft}, on which to perform the FFT. When N_{fft} exceeds the input frame size, the frame is zero-padded as needed. This parameter is enabled when you do not select **Inherit FFT length from estimation order**.

References Kay, S. M. *Modern Spectral Estimation: Theory and Application.* Englewood Cliffs, NJ: Prentice-Hall, 1988.

Yule-Walker Method

Marple, S. L., Jr., *Digital Spectral Analysis with Applications.* Englewood Cliffs, NJ: Prentice-Hall, 1987.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point

The output data type is the same as the input data type. To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Burg Method	Signal Processing Blockset
Covariance Method	Signal Processing Blockset
Levinson-Durbin	Signal Processing Blockset
Autocorrelation LPC	Signal Processing Blockset
Short-Time FFT	Signal Processing Blockset
Yule-Walker AR Estimator	Signal Processing Blockset
pyulear	Signal Processing Toolbox

See "Power Spectrum Estimation" on page 6-6 for related information.

Purpose   Count number of times signal crosses zero in a single time step

Library   Signal Operations

Description
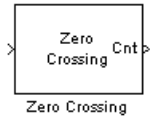
Zero Crossing

The Zero Crossing block concludes that a signal has passed through zero if it meets any of the following criteria, where x_i is the current signal value, x_{i-1} is the previous signal value, and so on:

- $x_i < 0$ and $x_{i-1} > 0$

- $x_i > 0$ and $x_{i-1} < 0$

- For some positive integer L, $x_i < 0$, $x_{i-l} = 0$, and $x_{i-L-1} > 0$, where $0 \leq l \leq L$.

- For some positive integer L, $x_i > 0$, $x_{i-l} = 0$, and $x_{i-L-1} < 0$, where $0 \leq l \leq L$.

For the first input value, x_{i-1} and x_{i-2} are zero. The block outputs the number of times the signal crosses zero in a single time step at the Cnt port.

The input to this block must be a real-valued, fixed-point or floating-point, single-channel signal. This means that sample-based signals must be scalars and frame-based signals must be vectors. The block produces an error if the input signal is multichannel.

Examples   The following example illustrates the behavior of the Zero Crossing block.

1 Create the following Simulink model.

Zero Crossing
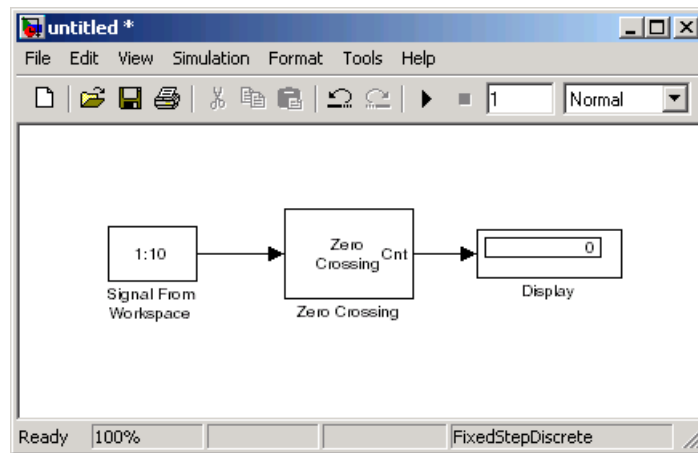
2 Use the Signal From Workspace block to create a frame-based signal. Set the parameters as follows:

- **Signal** = [-3:3]'

- **Sample time** = 1/7

- **Samples per frame** = 7

- **Form output after final data value by** = Cyclic repetition

The block outputs a single frame of the frame-based signal at the first time step, and identical frames at each additional time step.

3 Use the Zero Crossing block to detect the number of zero crossing in each time step. Use the default parameters.

4 Use the Display block to view the number of zero crossings.

5 To run the model for one time step, set the configuration parameters. Open the Configuration Parameters dialog box by selecting **Configuration Parameters** from the **Simulation** menu. In the **Solver** pane, set the parameters as follows:

- **Stop time** = 0

- **Type** = Fixed-step
- **Solver** = discrete (no continuous states)

6 Run the model.

Because the signal passes through zero once during the first time step, the Zero Crossing block finds one zero crossing as shown in the figure below.
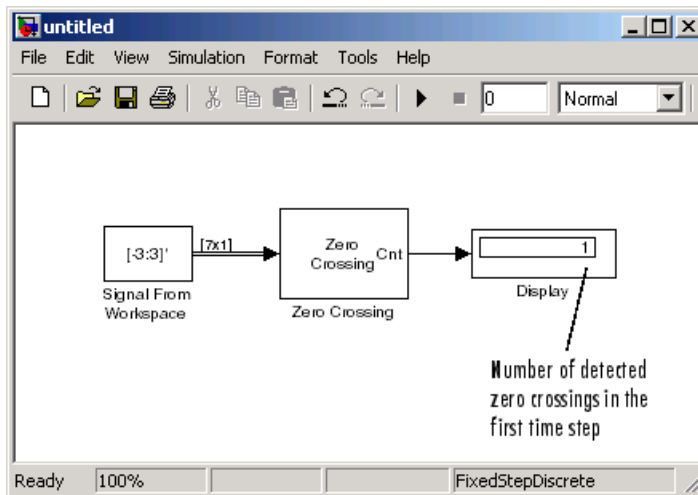
7 To run the model for two time steps, change the simulation **Stop time** to 1.

8 Run the model.

The Zero Crossing block remembers that the last value of the last frame was 3. Therefore, the signal passes through zero twice during the second time step. It passes through zero while going from 3 to -3, and it passes through zero again while going from -3 to 3. The Zero Crossing block finds two zero crossings in the second time step as shown in the figure below.

Zero Crossing

Dialog Box

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating-point
	• Single-precision floating-point
	• Fixed point (signed and unsigned)
	• 8-, 16-, and 32-bit signed integers
	• 8-, 16-, and 32-bit unsigned integers
Cnt	• 32-bit unsigned integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Hit Crossing   Simulink

Zero Pad

Purpose

Alter input dimensions by zero-padding (or truncating) rows and/or columns

Library

Signal Operations

Description

The Zero Pad block changes the dimensions of the input matrix from M_i-by-N_i to M_o-by-N_o by zero-padding or truncating along the columns, rows, or columns and rows. Use the **Pad along** parameter to specify the dimensions to change.

Using the **Pad signal at** parameter, you can choose to pad your input matrix at the end or the beginning of a row and/or column.

The **Number of output rows** and/or **Number of output columns** parameters refer to the dimensions of the output, M_o and N_o. You can set these parameters to User-specified or Next power of two. When you choose User-specified, enter a scalar value in the **Specified number of output rows** and/or **Specified number of output columns** parameters. When you choose Next power of two, the block pads the input matrix along the columns and/or rows until the length of the columns and/or rows is equal to a power of two. When the length of the input matrix's columns and/or rows is already equal to a power of two, the block does not pad the input matrix.

When you choose User-specified for the **Number of output rows** and/or **Number of output columns** parameters, you can specify a scalar value in the **Specified number of output rows** and/or **Specified number of output columns** parameters that truncates the size of your input matrix. The following options are available for the **Action when truncation occurs** parameter:

- None — Select this option when you do not want to be notified that the input matrix is truncated.

- Warning — Choose this option when you want a warning to be displayed in the MATLAB Command Window when the input matrix is truncated.

- Error — Click this option when you want an error dialog box to be displayed and the simulation terminated when the input matrix is truncated.

The behavior of the Pad block and Zero Pad block is identical, with the exception that the Pad block can pad the input matrix with values other than zero. See the Pad block reference page for more information on the behavior of the Pad block.

Example   In the model below, the 3-by-3 input is zero-padded along the column dimension to 5-by-3. The parameter settings in the Zero Pad block are

- **Pad signal at** =End
- **Pad along** = Columns
- **Number of output rows** = User-specified
- **Specified number of output rows** = 5
- **Action when truncation occurs:** None

The following figure shows the result of running the model.
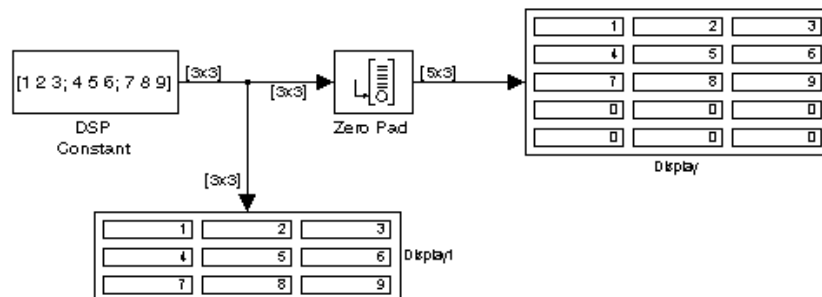
Zero Pad
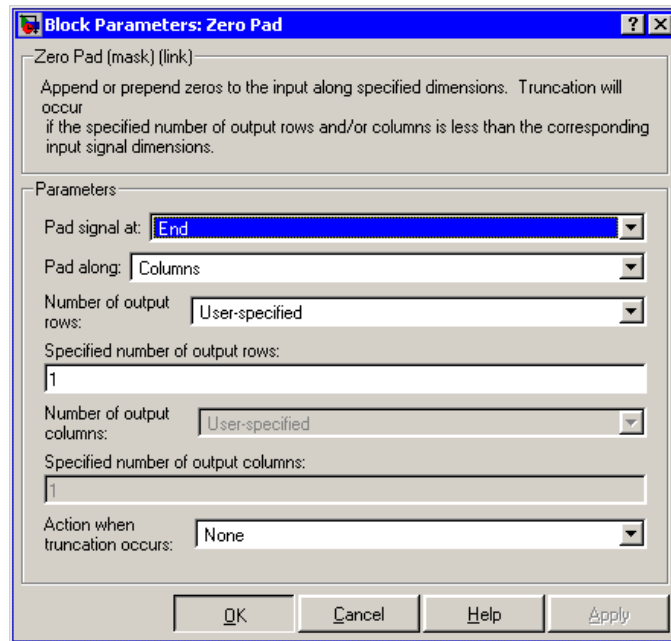
Pad signal at
> The input matrix can be padded at the beginning of the rows and/or columns or at the end of the rows and/or columns.

Pad along
> The direction along which to pad or truncate. Columns specifies that the *row* dimension should be changed to M_o. Rows specifies that the *column* dimension should be changed to N_o. Columns and rows specifies that both column and row dimensions should be changed. None disables padding and truncation and passes the input through to the output unchanged.

Number of output rows
> The total number of output rows can be User-specified or Next power of two. When you select User-specified, type a scalar value in the **Specified Number of output rows** parameter. When you select Next power of two, the block pads the columns

of the input matrix until the number of rows is equal to a power of two. When the number of rows is already equal to a power of two, the block does not pad the input matrix.

Specified number of output rows

The desired number of rows in the output, M_o. This parameter is enabled when you select Columns or Columns and rows in the **Pad along** menu and User-specified is chosen in the **Number of output rows** parameter.

Number of output columns

The total number of output columns. This parameter is enabled when you select Rows or Columns and rows in the **Pad along** menu. When you select User-specified, type a scalar value in the **Specified Number of output columns** parameter. When you select Next power of two, the block pads the rows of the input matrix until the number of columns is equal to a power of two. When the number of columns is already equal to a power of two, the block does not pad the input matrix.

Specified number of output columns

The desired number of columns in the output, N_o. This parameter is enabled when you select Rows or Columns and rows in the **Pad along** menu and User-specified is chosen in the **Number of output columns** parameter.

Action when truncation occurs

Choose None when you do not want to be notified that the input matrix is truncated. Select Warning to display a warning when the input matrix is truncated. Choose Error when you want an error dialog box to be displayed and the simulation terminated when the input matrix is truncated.

Zero Pad

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating-point • Single-precision floating-point • Fixed point (signed only) • Custom data types • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating-point • Single-precision floating-point • Fixed point (signed only) • Custom data types • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

To learn how to convert your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page 7-2.

See Also

Matrix Concatenation	Simulink
Pad	Signal Processing Blockset
Repeat	Signal Processing Blockset
Submatrix	Signal Processing Blockset

Upsample	Signal Processing Blockset
Variable Selector	Signal Processing Blockset

Functions — Alphabetical List

dsp_links

Purpose   Display library link information for blocks linked to the Signal
                   Processing Blockset

Syntax   dsp_links
                   dsp_links()

Description   The dsp_links function displays and returns library link information
                   for blocks linked to the Signal Processing Blockset libraries.

                   Signal Processing Blockset blocks can be obsolete, deprecated, or
                   current. Obsolete blocks are blocks that are no longer supported. They
                   might or might not work properly. Deprecated blocks are still supported
                   but are likely to become obsolete in a future release. Current blocks are
                   supported and represent the latest block functionality.

                   dsp_links() returns a structure with three elements for the current
                   model. Each element represents one of the three block categories and
                   contains a cell array of strings. Each string is the name of a library
                   block in the current model.

                   dsp_links(sys) returns a structure with three elements for the named
                   system.

See Also

                   liblinks   Signal Processing Blockset

Purpose Open the main Signal Processing Blockset library

Syntax dsplib

Description dsplib opens the current version of the main Signal Processing Blockset library.

dspstartup

Purpose   Configure the Simulink environment for signal processing systems

Syntax   `dspstartup`

Description `dspstartup` configures a number of Simulink environment parameters with settings appropriate for a typical DSP project. When the Simulink environment has successfully been configured, the function displays the following message in the command window.

```
Changed default Simulink settings for DSP systems (dspstartup.m).
```

To automatically configure the Simulink environment at startup, add a call to `dspstartup.m` from your `startup.m` file. If you do not have a `startup.m` file on your path, you can create one from the `startupsav.m` template in the `toolbox/local` directory.

To edit `startupsav.m`, simply replace the `load matlab.mat` command with a call to `dspstartup.m`, and save the file as `startup.m`. The result should look like this.

```
%STARTUP Startup file
%    This file is executed when MATLAB starts up,
%    if it exists anywhere on the path.
dspstartup;
```

For more information, see the description for the `startup` command in the MATLAB documentation and Configuring Simulink for Signal Processing Models in the Getting Started Signal Processing Blockset documentation.

The `dspstartup.m` script sets the following Simulink environment parameters. See "Model and Block Parameters" in the Using Simulink documentation for complete information about a particular setting.

Parameter	Setting
SingleTaskRate TransMsg	error
Solver	fixedstepdiscrete
SolverMode	SingleTasking
StartTime	0.0
StopTime	inf
FixedStep	auto
SaveTime	off
SaveOutput	off
AlgebraicLoopMsg	error
InvariantConstants	on

See Also

startup   MATLAB

liblinks

Purpose   Display library link information for blocks linked to the Signal Processing Blockset

Syntax

Description   Please see the command line help for `liblinks`. Type

```
help liblinks
```

in the MATLAB Command Window.

See Also

    `dsp_links`   Signal Processing Blockset

Purpose   Compute the number of samples of delay introduced by buffering and unbuffering operations

Syntax
```
d = rebuffer_delay(f,n,m)
d = rebuffer_delay(f,n,m,'singletasking')
```

Description   d = rebuffer_delay(f,n,m) returns the delay (in samples) introduced by the buffering and unbuffering blocks in multitasking operations, where f is the input frame size, n is the **Output buffer size** parameter setting, and m is the **Buffer overlap** parameter setting.

The blocks whose delay can be computed by rebuffer_delay are

- Buffer
- Unbuffer

d = rebuffer_delay(f,n,m,'singletasking') returns the delay (in samples) introduced by these blocks in single-tasking operations.

The table below shows the appropriate rebuffer_delay parameter values to use in computing delay for the two blocks.

Block	Parameter Values
Buffer	f = input frame size (f=1 for sample-based mode)
	n = **Output buffer size**
	m = **Buffer overlap**
Unbuffer	f = input frame size
	n = 1
	m = 0

See Also

Buffer	Signal Processing Blockset
Unbuffer	Signal Processing Blockset

Glossary

This glossary defines terms related to fixed-point data types and numbers. These terms may appear in some or all of the documents that describe products from The MathWorks that have fixed-point support.

arithmetic shift

Shift of the bits of a binary word for which the sign bit is recycled for each bit shift to the right. A zero is incorporated into the least significant bit of the word for each bit shift to the left. In the absence of overflows, each arithmetic shift to the right is equivalent to a division by 2, and each arithmetic shift to the left is equivalent to a multiplication by 2.

See also binary point, binary word, bit, logical shift, most significant bit

bias

Part of the numerical representation used to interpret a fixed-point number. Along with the slope, the bias forms the scaling of the number. Fixed-point numbers can be represented as

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

where the slope can be expressed as

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

See also fixed-point representation, fractional slope, integer, scaling, slope, [Slope Bias]

binary number

Value represented in a system of numbers that has two as its base and that uses 1's and 0's (bits) for its notation.

See also bit

binary point

Symbol in the shape of a period that separates the integer and fractional parts of a binary number. Bits to the left of the binary point are integer bits and/or sign bits, and bits to the right of the binary point are fractional bits.

See also binary number, bit, fraction, integer, radix point

binary point-only scaling

Scaling of a binary number that results from shifting the binary point of the number right or left, and which therefore can only occur by powers of two.

See also binary number, binary point, scaling

binary word

Fixed-length sequence of bits (1's and 0's). In digital hardware, numbers are stored in binary words. The way in which hardware components or software functions interpret this sequence of 1's and 0's is described by a data type.

See also bit, data type, word

bit

Smallest unit of information in computer software or hardware. A bit can have the value 0 or 1.

ceiling (round toward)

Rounding mode that rounds to the closest representable number in the direction of positive infinity. This is equivalent to the `ceil` mode in Fixed-Point Toolbox.

See also convergent rounding, floor (round toward), nearest (round toward), rounding, truncation, zero (round toward)

contiguous binary point

Binary point that occurs within the word length of a data type. For example, if a data type has four bits, its contiguous binary point must be understood to occur at one of the following five positions:

```
.0000
0.000
00.00
000.0
0000.
```

See also data type, noncontiguous binary point, word length

convergent rounding

Rounding mode that rounds to the nearest allowable quantized value. Numbers that are exactly halfway between the two nearest allowable quantized values are rounded up only if the least significant bit (after rounding) would be set to 0.

See also ceiling (round toward), floor (round toward), nearest (round toward), rounding, truncation, zero (round toward)

data type

Set of characteristics that define a group of values. A fixed-point data type is defined by its word length, its fraction length, and whether it is signed or unsigned. A floating-point data type is defined by its word length and whether it is signed or unsigned.

See also fixed-point representation, floating-point representation, fraction length, word length

data type override

Parameter in the Fixed-Point Settings interface that allows you to set the output data type and scaling of fixed-point blocks on a system or subsystem level.

See also data type, scaling

exponent

Part of the numerical representation used to express a floating-point or fixed-point number.

1. Floating-point numbers are typically represented as

$$real\text{-}world\ value\ =\ mantissa \times 2^{exponent}$$

2. Fixed-point numbers can be represented as

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

where the slope can be expressed as

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

The exponent of a fixed-point number is equal to the negative of the fraction length:

$$exponent = -1 \times fraction\ length$$

See also bias, fixed-point representation, floating-point representation, fraction length, fractional slope, integer, mantissa, slope

fixed-point representation

Method for representing numerical values and data types that have a set range and precision.

1. Fixed-point numbers can be represented as

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

where the slope can be expressed as

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

The slope and the bias together represent the scaling of the fixed-point number.

2. Fixed-point data types can be defined by their word length, their fraction length, and whether they are signed or unsigned.

See also bias, data type, exponent, fraction length, fractional slope, integer, precision, range, scaling, slope, word length

floating-point representation

Method for representing numerical values and data types that can have changing range and precision.

1. Floating-point numbers can be represented as

$$real\text{-}world\ value = mantissa \times 2^{exponent}$$

2. Floating-point data types are defined by their word length.

See also data type, exponent, mantissa, precision, range, word length

floor (round toward)

Rounding mode that rounds to the closest representable number in the direction of negative infinity.

See also ceiling (round toward), convergent rounding, nearest (round toward), rounding, truncation, zero (round toward)

fraction

Part of a fixed-point number represented by the bits to the right of the binary point. The fraction represents numbers that are less than one.

See also binary point, bit, fixed-point representation

fraction length

Number of bits to the right of the binary point in a fixed-point representation of a number.

See also binary point, bit, fixed-point representation, fraction

fractional slope

Part of the numerical representation used to express a fixed-point number. Fixed-point numbers can be represented as

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

where the slope can be expressed as

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

The term *slope adjustment* is sometimes used as a synonym for fractional slope.

See also bias, exponent, fixed-point representation, integer, slope

guard bits

Extra bits in either a hardware register or software simulation that are added to the high end of a binary word to ensure that no information is lost in case of overflow.

See also binary word, bit, overflow

integer

1. Part of a fixed-point number represented by the bits to the left of the binary point. The integer represents numbers that are greater than or equal to one.

2. Also called the "stored integer." The raw binary number, in which the binary point is assumed to be at the far right of the word. The integer is part of the numerical representation used to express a fixed-point number. Fixed-point numbers can be represented as

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

or

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

where the slope can be expressed as

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

See also bias, fixed-point representation, fractional slope, integer, real-world value, slope

integer length

Number of bits to the left of the binary point in a fixed-point representation of a number.

See also binary point, bit, fixed-point representation, fraction length, integer

least significant bit (LSB)

Bit in a binary word that can represent the smallest value. The LSB is the rightmost bit in a big-endian-ordered binary word. The weight of the LSB is related to the fraction length according to

$$weight\ of\ LSB\ =\ 2^{-fraction\ length}$$

See also big-endian, binary word, bit, most significant bit

logical shift

Shift of the bits of a binary word, for which a zero is incorporated into the most significant bit for each bit shift to the right and into the least significant bit for each bit shift to the left.

See also arithmetic shift, binary point, binary word, bit, most significant bit

mantissa

Part of the numerical representation used to express a floating-point number. Floating-point numbers are typically represented as

$$real\text{-}world\ value\ =\ mantissa \times 2^{exponent}$$

See also exponent, floating-point representation

most significant bit (MSB)

Bit in a binary word that can represent the largest value. The MSB is the leftmost bit in a big-endian-ordered binary word.

See also binary word, bit, least significant bit

nearest (round toward)

Rounding mode that rounds to the closest representable number, with the exact midpoint rounded to the closest representable number in the direction of positive infinity. This is equivalent to the `nearest` mode in Fixed-Point Toolbox.

See also ceiling (round toward), convergent rounding, floor (round toward), rounding, truncation, zero (round toward)

noncontiguous binary point

Binary point that is understood to fall outside the word length of a data type. For example, the binary point for the following 4-bit word is understood to occur two bits to the right of the word length,

```
0000_ _.
```

thereby giving the bits of the word the following potential values:

$$2^5 2^4 2^3 2^2 \_\ \_.$$

See also binary point, data type, word length

one's complement representation

Representation of signed fixed-point numbers. Negating a binary number in one's complement requires a bitwise complement. That is, all 0's are flipped to 1's and all 1's are flipped to 0's. In one's complement notation there are two ways to represent zero. A binary word of all 0's represents "positive" zero, while a binary word of all 1's represents "negative" zero.

See also binary number, binary word, sign/magnitude representation, signed fixed-point, two's complement representation

overflow

Situation that occurs when the magnitude of a calculation result is too large for the range of the data type being used. In many cases you can choose to either saturate or wrap overflows.

See also saturation, wrapping

padding

Extending the least significant bit of a binary word with one or more zeros.

See also least significant bit

precision

1. Measure of the smallest numerical interval that a fixed-point data type and scaling can represent, determined by the value of the number's least significant bit. The precision is given by the slope, or the number of fractional bits. The term *resolution* is sometimes used as a synonym for this definition.

2. Measure of the difference between a real-world numerical value and the value of its quantized representation. This is sometimes called quantization error or quantization noise.

See also data type, fraction, least significant bit, quantization, quantization error, range, slope

Q format

Representation used by Texas Instruments to encode signed two's complement fixed-point data types. This fixed-point notation takes the form

Qm.n

where

- *Q* indicates that the number is in Q format.

- *m* is the number of bits used to designate the two's complement integer part of the number.

- *n* is the number of bits used to designate the two's complement fractional part of the number, or the number of bits to the right of the binary point.

In Q format notation, the most significant bit is assumed to be the sign bit.

See also binary point, bit, data type, fixed-point representation, fraction, integer, two's complement

quantization

Representation of a value by a data type that has too few bits to represent it exactly.

See also bit, data type, quantization error

quantization error

Error introduced when a value is represented by a data type that has too few bits to represent it exactly, or when a value is converted from one data type to a shorter data type. Quantization error is also called quantization noise.

See also bit, data type, quantization

radix point

Symbol in the shape of a period that separates the integer and fractional parts of a number in any base system. Bits to the left of the radix point are integer and/or sign bits, and bits to the right of the radix point are fraction bits.

See also binary point, bit, fraction, integer, sign bit

range

Span of numbers that a certain data type can represent.

See also data type, precision

real-world value

Stored integer value with fixed-point scaling applied. Fixed-point numbers can be represented as

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

or

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

where the slope can be expressed as

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

See also integer

resolution

See **precision**

rounding

Limiting the number of bits required to express a number. One or more least significant bits are dropped, resulting in a loss of precision. Rounding is necessary when a value cannot be expressed exactly by the number of bits designated to represent it.

See also bit, ceiling (round toward), convergent rounding, floor (round toward), least significant bit, nearest (round toward), precision, truncation, zero (round toward)

saturation

Method of handling numeric overflow that represents positive overflows as the largest positive number in the range of the data type being used, and negative overflows as the largest negative number in the range.

See also overflow, wrapping

scaling

1. Format used for a fixed-point number of a given word length and signedness. The slope and bias together form the scaling of a fixed-point number.

2. Changing the slope and/or bias of a fixed-point number without changing the stored integer.

See also bias, fixed-point representation, integer, slope

shift

Movement of the bits of a binary word either toward the most significant bit ("to the left") or toward the least significant bit ("to the right"). Shifts to the right can be either logical, where the spaces emptied at the front of the word with each shift are filled in with zeros, or arithmetic, where the word is sign extended as it is shifted to the right.

See also arithmetic shift, logical shift, sign extension

sign bit

Bit (or bits) in a signed binary number that indicates whether the number is positive or negative.

See also binary number, bit

sign extension

Addition of bits that have the value of the most significant bit to the high end of a two's complement number. Sign extension does not change the value of the binary number.

See also binary number, guard bits, most significant bit, two's complement representation, word

sign/magnitude representation

Representation of signed fixed-point or floating-point numbers. In sign/magnitude representation, one bit of a binary word is always the dedicated sign bit, while the remaining bits of the word encode the magnitude of the number. Negation using sign/magnitude representation consists of flipping the sign bit from 0 (positive) to 1 (negative), or from 1 to 0.

Glossary-11

See also binary word, bit, fixed-point representation, floating-point representation, one's complement representation, sign bit, signed fixed-point, two's complement representation

signed fixed-point

Fixed-point number or data type that can represent both positive and negative numbers.

See also data type, fixed-point representation, unsigned fixed-point

slope

Part of the numerical representation used to express a fixed-point number. Along with the bias, the slope forms the scaling of a fixed-point number. Fixed-point numbers can be represented as

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

where the slope can be expressed as

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

See also bias, fixed-point representation, fractional slope, integer, scaling, [Slope Bias]

slope adjustment

See **fractional slope**

[Slope Bias]

Representation used to define the scaling of a fixed-point number.

See also bias, scaling, slope

stored integer

See **integer**

trivial scaling

Scaling that results in the real-world value of a number being simply equal to its stored integer value:

$$real\text{-}world\ value\ =\ integer$$

In [Slope Bias] representation, fixed-point numbers can be represented as

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

In the trivial case, slope = 1 and bias = 0.

In terms of binary point-only scaling, the binary point is to the right of the least significant bit for trivial scaling, meaning that the fraction length is zero:

$$real\text{-}world\ value\ =\ integer \times 2^{-fraction\ length}\ =\ integer \times 2^{0}$$

Scaling is always trivial for pure integers, such as `int8`, and also for the true floating-point types `single` and `double`.

See also bias, binary point, binary point-only scaling, fixed-point representation, fraction length, integer, least-significant bit, scaling, slope, [Slope Bias]

truncation

Rounding mode that drops one or more least significant bits from a number.

See also ceiling (round toward), convergent rounding, floor (round toward), nearest (round toward), rounding, zero (round toward)

two's complement representation

Common representation of signed fixed-point numbers. Negation using signed two's complement representation consists of a translation into one's complement followed by the binary addition of a one.

See also binary word, one's complement representation, sign/magnitude representation, signed fixed-point

unsigned fixed-point

Fixed-point number or data type that can only represent numbers greater than or equal to zero.

See also data type, fixed-point representation, signed fixed-point

Glossary-13

word
> Fixed-length sequence of binary digits (1's and 0's). In digital hardware, numbers are stored in words. The way hardware components or software functions interpret this sequence of 1's and 0's is described by a data type.

> *See also* binary word, data type

word length
> Number of bits in a binary word or data type.

> *See also* binary word, bit, data type

wrapping
> Method of handling overflow. Wrapping uses modulo arithmetic to cast a number that falls outside of the representable range the data type being used back into the representable range.

> *See also* data type, overflow, range, saturation

zero (round toward)
> Rounding mode that rounds to the closest representable number in the direction of zero. This is equivalent to the `fix` mode in Fixed-Point Toolbox.

> *See also* ceiling (round toward), convergent rounding, floor (round toward), nearest (round toward), rounding, truncation

Index