DSP Blockset

For Use with Simulink®

Modeling

Simulation

Implementation

User's Guide

Version 5

The MathWorks

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.

DSP Blockset User's Guide

Printing History:			
	April 1995	First printing	DSP Blockset 1.0
	May 1997	Second printing	DSP Blockset 2.0
	January 1998	Third printing	DSP Blockset 2.2 (Release 10)
	January 1999	Fourth printing	DSP Blockset 3.0 (Release 11)
	November 2000	Fifth printing	DSP Blockset 4.0 (Release 12)
	June 2001	Online only	DSP Blockset 4.1 (Release 12.1)
	July 2002	Sixth printing	DSP Blockset 5.0 (Release 13)

Contents

1 Introduction

2 Simulink Review

3 Working with Signals

4 Filters

5 Transforms

6 Statistics, Estimation, and Linear Algebra

7 Block Reference

8 Function Reference

A Data Type Support

B Code Generation Support

C Configuring Simulink for DSP Systems

1

Introduction

Welcome to the DSP Blockset

Welcome to the DSP Blockset, the premier tool for digital signal processing (DSP) algorithm simulation and code generation. This section contains the following topics, which help introduce you to the DSP Blockset:

- "What Is the DSP Blockset?" on page 1-3
- "What Is in the DSP Blockset?" on page 1-6
- "Related Products" on page 1-8
- "Getting Help with the DSP Blockset" on page 1-10
- "Technical Conventions" on page 1-13
- "Typographical Conventions" on page 1-16
- "Installing DSP Blockset" on page 1-17

The DSP Blockset brings the full power of Simulink® to DSP system design and prototyping by providing key DSP algorithms and components in the adaptable block format of Simulink. From buffers to linear algebra solvers, from dyadic filter banks to parametric estimators, the blockset gives you all the core components to rapidly and efficiently assemble complex DSP systems.

Use the DSP Blockset and Simulink to develop your DSP concepts, and to efficiently revise and test until your design is production-ready. Use the DSP Blockset together with the Real-Time Workshop® to automatically generate code for real-time execution on DSP hardware.

We hope you enjoy using the DSP Blockset, and we look forward to hearing your comments and suggestions.

`support@mathworks.com`	Technical support
`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports

Visit the MathWorks Web site at `www.mathworks.com` for complete contact information.

What Is the DSP Blockset?

The DSP Blockset is a collection of block libraries for use with the Simulink dynamic system simulation environment.

The DSP Blockset libraries are designed specifically for digital signal processing (DSP) applications, and include key operations such as classical, multirate, and adaptive filtering, matrix manipulation and linear algebra, statistics, time-frequency transforms, and more.

Key Features

The DSP Blockset extends the Simulink environment by providing core components and algorithms for DSP systems. You can use blocks from the DSP Blockset in the same way that you would use any other Simulink blocks, combining them with blocks from other libraries to create sophisticated DSP systems.

A few of the important features are described in the following sections:

- "Frame-Based Operations" on page 1-3
- "Matrix Support" on page 1-4
- "Adaptive and Multirate Filtering" on page 1-4
- "Statistical Operations" on page 1-5
- "Linear Algebra" on page 1-5
- "Parametric Estimation" on page 1-5
- "Real-Time Code Generation" on page 1-5

Frame-Based Operations

Most real-time DSP systems optimize throughput rates by processing data in "batch" or "frame-based" mode, where each batch or frame is a collection of consecutive signal samples that have been buffered into a single unit. By propagating these multisample frames instead of the individual signal samples, the DSP system can best take advantage of the speed of DSP algorithm execution, while simultaneously reducing the demands placed on the data acquisition (DAQ) hardware.

The DSP Blockset delivers this same high level of performance for both simulation and code generation by incorporating frame-processing capability into all of its blocks. A completely frame-based model can run several times faster than the same model processing sample-by-sample; faster still if data sources are frame based.

See “Sample Rates and Frame Rates” on page 3-16 for more information.

Matrix Support

The DSP Blockset takes full advantage of the matrix format of Simulink. Some typical uses of matrices in DSP simulations are

- *General two-dimensional array*

  A matrix can be used in its traditional mathematical capacity, as a simple structured array of numbers. Most blocks for general matrix operations are found in the Matrices and Linear Algebra library.
- *Factored submatrices*

  A number of the matrix factorization blocks in the Matrix Factorizations library store the submatrix factors (i.e., lower and upper submatrices) in a single compound matrix. See the LDL Factorization and LU Factorization blocks for examples.
- *Multichannel frame-based signal*

  The standard format for multichannel frame-based data is a matrix containing each channel’s data in a separate column. A matrix with three columns, for example, contains three channels of data, one frame per channel. The number of rows in such a matrix is the number of samples in each frame.

See the following sections for more information about working with matrices:

- “Multichannel Signals” on page 3-11
- “Creating Signals” on page 3-33
- “Constructing Signals” on page 3-42
- “Importing Signals” on page 3-62

Adaptive and Multirate Filtering

The Adaptive Filters and Multirate Filters libraries provide key tools for the construction of advanced DSP systems. Adaptive filter blocks are parameterized to support the rapid tailoring of DSP algorithms to application-specific environments, and effortless “what if” experimentation.

The multirate filtering algorithms employ polyphase implementations for efficient simulation and real-time code execution.

See “Multirate Filters” on page 4-32 and “Adaptive Filters” on page 4-34 for more information.

Statistical Operations

Use the blocks in the Statistics library for basic statistical analysis. These blocks calculate measures of central tendency and spread (e.g., mean, standard deviation, and so on), as well as the frequency distribution of input values (histograms).

See “Statistics” on page 6-3 for more information.

Linear Algebra

The Matrices and Linear Algebra library provides a wide variety of matrix factorization methods, and equation solvers based on these methods. The popular Cholesky, LU, LDL, and QR factorizations are all available.

See “Linear Algebra” on page 6-7 for more information.

Parametric Estimation

The Parametric Estimation library provides a number of methods for modeling a signal as the output of an AR system. The methods include the Burg AR Estimator, Covariance AR Estimator, Modified Covariance AR Estimator, and Yule-Walker AR Estimator, which allow you to compute the AR system parameters based on forward error minimization, backward error minimization, or both.

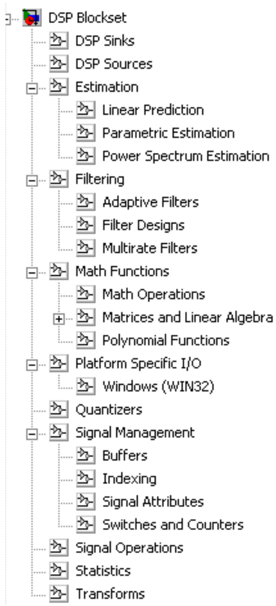
Real-Time Code Generation

You can also use the separate Real-Time Workshop product to generate optimized, compact, ANSI C code for models containing blocks from the DSP Blockset.

See Appendix B, “Code Generation Support” for more information.

What Is in the DSP Blockset?

The DSP Blockset contains a collection of blocks organized in a set of nested libraries. The best way to explore the blockset is to expand the **DSP Blockset** entry in the Simulink Library Browser. The fully expanded library list is shown below.



See the Simulink documentation for complete information about the Library Browser. To access the blockset through its own window (rather than through the Library Browser), type

```
dsplib
```

in the command window. Double-click on any library in the window to display its contents. The Demos block opens the MATLAB® Demos utility with the DSP Blockset demos selected.

Double-click on a demo in the list to open that model, and select **Start** from the model window's **Simulation** menu to run it.

For a complete listing, by category, of all the blocks in the DSP Blockset, see the online block reference, "Blocks — By Category."

Related Products

The MathWorks provides several products that are especially relevant to the kinds of tasks you can perform with the DSP Blockset.

For more information about any of these products, see either

- The online documentation for that product if it is installed or if you are reading the documentation from the CD
- The MathWorks Web site, at `www.mathworks.com`; see the "products" section

Note The toolboxes listed below all include functions that extend the capabilities of MATLAB. The blocksets all include blocks that extend the capabilities of Simulink. The DSP Blockset requires MATLAB, Simulink, and the Signal Processing Toolbox.

Product	Description
Communications Blockset	Design and simulate communication systems
Embedded Target for the TI C6000™ DSP Platform	Deploy and validate DSP designs on Texas Instruments C6000 digital signal processors
Filter Design Toolbox	Design and analyze advanced floating-point and fixed-point filters
MATLAB Link for Code Composer Studio™ Development Tools	Use MATLAB with RTDX™-enabled Texas Instruments digital signal processors
Real-Time Workshop	Generate C code from Simulink models
Signal Processing Toolbox	Perform signal processing, analysis, and algorithm development

Product	Description
Simulink	Design and simulate continuous- and discrete-time systems
Stateflow	Design and simulate event-driven systems

Getting Help with the DSP Blockset

The following sections provide information on how to get help with the DSP Blockset:

- "Using This Guide" on page 1-10
- "Getting Help Online" on page 1-10
- "How to Run Examples in the MATLAB Help Browser" on page 1-12

Using This Guide

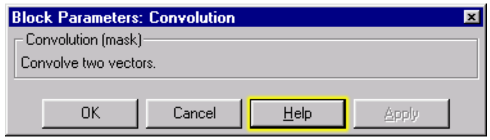
This guide contains tutorial sections that are designed to help you become familiar with using Simulink and the DSP Blockset, as well as a reference section for finding detailed information on particular blocks in the blockset:

- Read Chapter 2, "Simulink Review" to get an overview of fundamental Simulink and DSP Blockset concepts. Also see the Simulink documentation for more information on the Simulink environment.
- Read Chapter 3, "Working with Signals" for details on key operations common to many signal processing tasks.
- Read the following chapters for discussions of how to implement various DSP operations:
  - Chapter 4, "Filters"
  - Chapter 5, "Transforms"
  - Chapter 6, "Statistics, Estimation, and Linear Algebra"
- See the online block reference, "Blocks — By Category" for a description of each block's operation, parameters, and characteristics.

Use this guide in conjunction with the software to learn about the powerful features that the DSP Blockset provides.

Getting Help Online

There are a number of easy ways to get help on the DSP Blockset while you're working at the computer, as summarized in the following table.

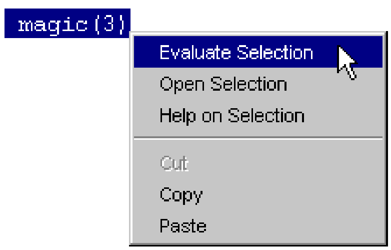
Where to Get Online Help	How to Get Online Help
Block Help	Click the **Help** button in any block dialog box to view the online reference documentation for that block. 
Simulink Library Browser	Right-click on a block to access the help for that block.
Help browser	Select **Full Product Family Help** from the **Help** menu, or type `doc` or `helpdesk` at the command line to display the Help browser. Select **DSP Blockset** in the **Contents** pane.
Command Line	Type `doc('block name')` at the command line to access the help for a block with the name `block name`. Spaces and capitalization in the block name are ignored.
Help Desk (remote)	Use a Web browser or the Help browser to connect to the MathWorks Web site at `www.mathworks.com`. Follow the **Documentation** link on the **Support** Web page for remote access to the documentation.
Release Information	Select **Full Product Family Help** from the **Help** menu, or type `whatsnew` at the MATLAB command line and select the DSP Blockset Release Notes from the **Contents** pane of the Help browser. The Release Notes contain information about new features and recent changes to the version of the DSP Blockset that you are using.

How to Run Examples in the MATLAB Help Browser

When viewing online reference pages in the MATLAB Help browser, you can easily run example code and example models without retyping code or reconstructing models. To learn how to access help in the Help browser, see the previous section, "Getting Help Online."

Running Example Code in the MATLAB Help Browser Without Retyping It.

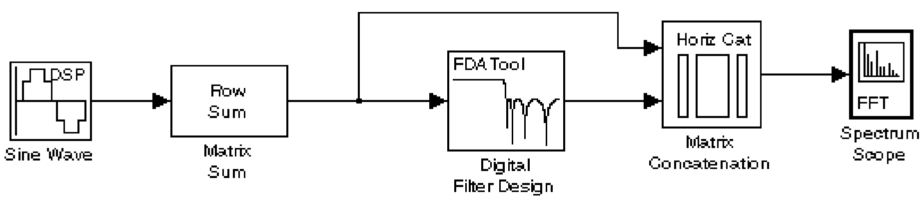
1. Highlight the code in the MATLAB Help browser.
2. Right-click on the selection.
3. Choose **Evaluate Selection** from the drop-down menu.



Running Example Models with the Click of the Mouse.

1. Click on indicated links in the MATLAB Help browser to bring up example models.
2. Run the models as you would run any other model. *Note the models may create new variables in your workspace*.

Open the following model by clicking here in the MATLAB Help browser.



Technical Conventions

The following sections provides a brief overview of the technical conventions used in this guide, and provides pointers to more detailed information:

- “Signal Dimension Nomenclature”
- “Frame-Based Signal Nomenclature” on page 1-14
- “Sampling Nomenclature” on page 1-15

Signal Dimension Nomenclature

The DSP Blockset fully supports the Simulink matrix format, which is described in “Working with Signals” in the Simulink documentation. The nomenclature used for vectors and matrices in the DSP Blockset is described below.

Matrices. Any mention of a *matrix* in the DSP Blockset is a reference to a *Simulink matrix*. A Simulink matrix is the same as a MATLAB matrix, a two-dimensional (2-D) array of values, organized as rows and columns. As in MATLAB, a matrix can be indexed by one or two values. The size of a matrix is described by the *number of rows* M and the *number of columns* N. In the DSP Blockset, matrix size is usually denoted by the compact expression M-by-N or M×N, and occasionally by the MATLAB notation `[M N]`.

For instance, a 2-by-3 matrix, like matrix *u* below, has two rows and three columns:

$$u = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

This matrix can be represented in MATLAB notation as

```
u = [1 2 3;4 5 6]      % A 2-by-3 matrix
```

In the online help, matrix elements are indexed using either subscript notation or MATLAB notation. For example, u_{23} and `u(2,3)` both refer to the element in the third column of the second row. The *number of channels* in a frame-based matrix is the number of columns, N. More information about matrices can be found in “Multichannel Signals” on page 3-11.

Vectors. The DSP Blockset refers to three different kinds of vectors:

- Row vector — A 1-by-N matrix

  $$\begin{bmatrix} a_1 & a_2 & \cdots & a_n \end{bmatrix}$$

- Column vector — An M-by-1 matrix

  $$\begin{bmatrix} a_1 \\ a_2 \\ M \\ a_n \end{bmatrix}$$

- One-dimensional (1-D) vector — A *1-D vector* is also known as a *Simulink vector* or *unoriented vector*. A 1-D vector is a one-dimensional array of values, an ordered list that has no row or column orientation. There is no MATLAB equivalent for 1-D vectors since all MATLAB vectors have either a row or column orientation. Most blocks in the DSP Blockset treat 1-D vectors as column vectors.

The *size* or *length* of a vector is the number of elements that it contains (M for a column vector or N for a row vector).

Arrays. The *number of pages*, P, of a three-dimensional array in the MATLAB workspace refers to the size of its third dimension:

```
A(:,:,1) = [1 2 3;4 5 6]   % The first page of a 3-page array
A(:,:,2) = [7 8 9;0 1 2]   % The second page
A(:,:,3) = [3 4 5;6 7 8]   % The last page
```

Array size is frequently denoted by the compact expression M-by-N-by-P or M×N×P.

Frame-Based Signal Nomenclature

A frame of data is a collection of sequential samples from a single channel. In Simulink, a length-M frame of data is represented by an M-by-1 matrix (column vector). A multichannel signal with N channels and M samples per frame is represented as an M-by-N matrix. See “Multichannel Signals” on page 3-11 for more about multichannel signals.

Signals in Simulink can be either frame-based or sample-based. You can typically specify the frame status (frame-based or sample-based) of any signal that you generate using a source block (from the DSP Sources library). Most other DSP blocks generally preserve the frame status of an input signal, but some do not. See “Creating Signals” on page 3-33 for more information.

Sampling Nomenclature

Important sampling-related notational conventions are listed in “Sample Rates and Frame Rates” on page 3-16.

Typographical Conventions

This guide uses some or all of these conventions.

Item	Convention	Example
Example code	`Monospace` font	To assign the value 5 to `A`, enter `A = 5`
Function names, syntax, filenames, directory/folder names, and user input	`Monospace` font	The `cos` function finds the cosine of each array element. Syntax line example is `MLGetVar ML_var_name`
Buttons and keys	**Boldface** with book title caps	Press the **Enter** key.
Literal strings (in syntax descriptions in reference chapters)	**`Monospace bold`** for literals	`f = freqspace(n,'`**`whole`**`')`
Mathematical expressions	*Italics* for variables Standard text font for functions, operators, and constants	This vector represents the polynomial $p = x^2 + 2x + 3$.

MATLAB output	`Monospace font`	MATLAB responds with
		`A =`
		`5`

Menu and dialog box titles	**Boldface** with book title caps	Choose the **File Options** menu.
New terms and for emphasis	*Italics*	An *array* is an ordered collection of information.
Omitted input arguments	(...) ellipsis denotes all of the input/output arguments from preceding syntaxes.	`[c,ia,ib] = union(...)`
String variables (from a finite list)	*`Monospace italics`*	`sysc = d2c(sysd,'`*`method`*`')`

Installing DSP Blockset

The DSP Blockset follows the same installation procedure as the MATLAB toolboxes. See the MATLAB Installation documentation for your platform.

2

Simulink Review

This chapter will help you get started building DSP models with Simulink and the DSP Blockset. It contains the following sections:

What Is Simulink?

Simulink is an environment for simulating dynamic systems. It provides a modeling and simulation "foundation" on which you can build digital signal processing applications. All of the blocks in the DSP Blockset are designed for use together with the blocks in the Simulink libraries.

This section includes the following topics:

- "Starting Simulink"
- "Getting Started with Simulink" on page 2-6
- "Learning More About Simulink" on page 2-11

Starting Simulink

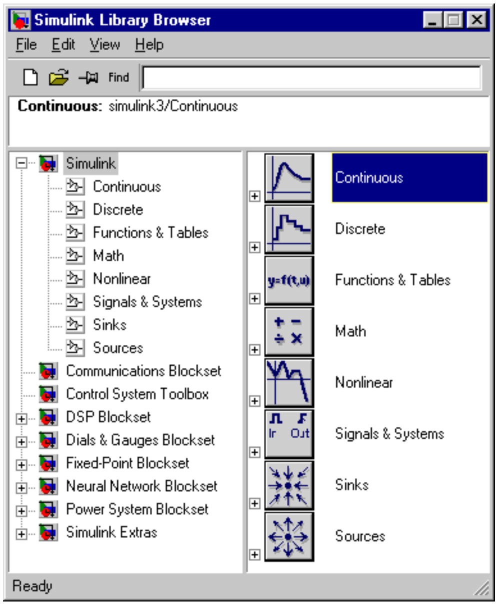
To start Simulink, click the  icon in the MATLAB toolbar, or type

```
simulink
```

at the command line.

Simulink on PC Platforms

On PC platforms, the Simulink Library Browser opens when you start Simulink. The left pane contains a list of all of the blocksets that you currently have installed.

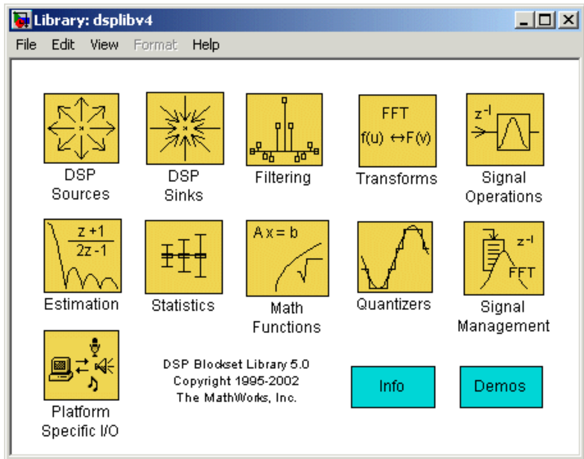


The first item in the list is the Simulink blockset itself, which is already expanded to show the available Simulink libraries. Click the ⊞ symbol to the left of any blockset name to expand the hierarchical list and display that blockset's libraries within the browser.

See the Simulink documentation for a complete description of the Library Browser.

Simulink on UNIX Platforms

On UNIX platforms, the Simulink window below opens when you start Simulink. To view other installed blocksets, double-click the **Blocksets & Toolboxes** button.



The following tutorial makes use of the Simulink Library Browser, available only on PC platforms. If you are working on a UNIX platform, instead of clicking the ⊞ symbol in the Library Browser to open a library, simply double-click the appropriate library in the main Simulink or DSP Blockset windows. To open the DSP Blockset window from the MATLAB command line, type `dsplib`.

The Simulink Libraries

The eight libraries in the Simulink window contain all of the basic elements you need to construct a model. Look here for basic math operations, switches, connectors, simulation control elements, and other items that do not have a specific DSP orientation.

To create a new model, select **New** from the Simulink **File** menu or press **Ctrl+N**. Then simply drag a block from one of the Simulink libraries into the new model window to begin building a system.

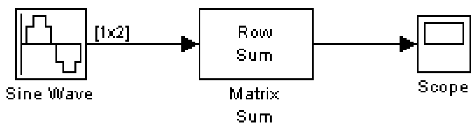
Getting Started with Simulink

If you have never used Simulink before, take some time to get acquainted with its features. You can begin by learning the two basic stages in model construction, discussed in the following sections:

- “Model Definition”
- “Model Simulation” on page 2-8

Model Definition

Simulink is a *model definition* environment. You define a model by creating a block diagram that represents the computations of your system or application. Try building a simple model that adds two sine waves and displays the result.



1 Type `dspstartup` at the MATLAB command line to configure Simulink for DSP simulation (optional).

   One of the things that `dspstartup` does is set the **Stop time** value in the **Simulation parameters** dialog box to `inf` for all new models. The `inf` setting instructs Simulink to run the model until you click the simulation stop button. You can access this dialog box and enter a different **Stop time** value by selecting **Simulation parameters** from the model window’s **Simulation** menu.

2 Start Simulink by clicking the  button in the MATLAB toolbar. The Library Browser appears.

3 Select **New > Model** from the **File** menu in the Library Browser. A new model window appears on your screen.

4 Add a Sine Wave block to the model.

   a In the Library Browser, click the ⊞ symbol next to **DSP Blockset** to expand the hierarchical list of DSP libraries.

b In the expanded list, click **DSP Sources** to view the blocks in the DSP Sources library.

c Drag the Sine Wave block into the new model window.

5 Add a Matrix Sum block to the model.

a Click the ⊞ symbol next to **Math Functions** to expand the Math Functions library.

b Click the ⊞ symbol next to **Matrices and Linear Algebra** to expand the Matrices and Linear Algebra sublibrary.

c In the expanded list, click **Matrix Operations** to view the blocks in the Matrix Operations library.

d Drag the Matrix Sum block into the model window.

6 Add a Scope block to the model.

a Click **Sinks** (in the Simulink tree) to view the blocks in the Simulink Sinks library.

b Drag the Scope block from the Sinks library into the model window. (The Simulink Scope block is the same as the Time Scope block in the DSP Sinks library.)

7 Connect the blocks.

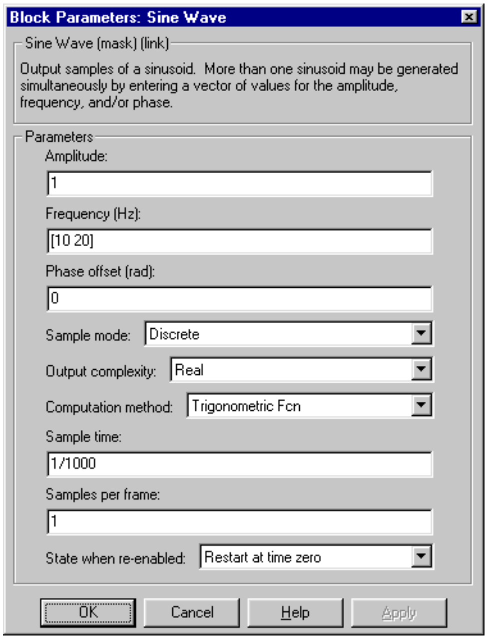
a Position the pointer near the output port of the Sine Wave block. Hold down the mouse button (the left button for a multibutton mouse) and drag the line that appears until it touches the input port of the Matrix Sum block. Release the mouse button.

b Using the same technique, connect the output of the Matrix Sum block to the input port of the Scope block.

8 Set the block parameters.

a Double-click on the Sine Wave block. The dialog box that appears allows you to set the block's parameters. *Parameters* are defining values that tell the block how to operate.

For this example, configure the block to generate a 10 Hz sine wave and a 20 Hz sine wave by entering `[10 20]` for the **Frequency** parameter. Both sinusoids will have the default amplitude of `1` and phase of `0` specified by the **Amplitude** and **Phase offset** parameters. They also both share the default sample period of 0.001 second specified by the **Sample time** parameter, which represents a sample rate of 1000 Hz.



Close the dialog box by clicking on the **OK** button or by pressing **Enter** on the keyboard.

b Double-click on the Matrix Sum block. Select **Rows** from the **Sum along** parameter, and close the dialog box.

You can now move on to the model simulation phase.

Model Simulation

Simulink is also a *model simulation* environment. You can run the simulation block diagram that you have built to see how the system behaves. To do this:

1 Select **Signal dimensions** from the **Format** menu (optional). The symbol "`[1x2]`" appears on the output line from Sine Wave indicating that the output is a 1-by-2 matrix.

At each sample time, the output matrix contains one sample from each of the two sinusoids. The Matrix Sum block adds the two matrix elements together to produce a scalar output. Thus, the input to the Scope block is the point-by-point sum of the two sinusoids.

2 Double-click on the Scope block if the Scope window is not already open on your screen. The scope window appears.

3 Select **Start** from the **Simulation** menu in the block diagram window. The signal containing the summed 10 Hz and 20 Hz component sinusoids is plotted on the scope.

4 Adjust the Scope block's display.

a While the simulation is running, right-click on the *y*-axis of the scope and select **Autoscale**. The vertical range of the scope is adjusted to better fit the signal.

b Click the **Properties** button on the scope, , and enter `0.1` for **Time range**. This resizes the scope's time axis to display only one cycle of the signal.

5 Vary the Sine Wave block parameters.

a While the simulation is running, double-click on the Sine Wave block to open it.

b Change the frequencies of the two sinusoids. Try entering `[1 5]` or `[100 400]` in the **Frequency** field. Click **Apply** after entering each new value, and observe the changes on the scope.

Note that the sample rate of both sinusoids is 1000 Hz, so aliasing will occur for sinusoid frequencies above 500 Hz. You can increase the sample rate by entering a smaller value in the Sine Wave block's **Sample time** parameter. This parameter is not tunable (see below), so you will need to stop the simulation before making any adjustment.

6 Select **Stop** from the **Simulation** menu to stop the simulation.

Tunable Parameters

Many parameters *cannot* be changed while a simulation is running. This is usually the case for parameters that directly or indirectly alter a signal's dimensions or sample rate. There are some parameters, however, such as the Sine Wave **Frequency** parameter, that you can change or *tune* while a simulation runs. In the online block reference pages, these parameters are marked "Tunable" in the parameter descriptions.

How to Tune Tunable Parameters. To tune a tunable parameter during a simulation, double-click the block to open its **Block Parameters** dialog, change any tunable parameters to the desired settings, and then click **OK** or **Apply**. The simulation continues to run, but with the new parameter settings.

Tunability in Simulation, Accelerator, and External Mode. Block parameters can be tunable in simulation, in the Simulink Performance Tools Accelerator, and in Real-Time Workshop external mode. When a parameter is marked "Tunable" in a reference page, it is tunable only in simulation, unless indicated otherwise.

Running a Simulation from an M-File

You can also modify and run a Simulink simulation from within a MATLAB M-file. By doing this, you can automate the variation of model parameters to explore a large number of simulation conditions rapidly and efficiently. For information on how to do this, see "Delay and Latency" on page 3-85 and "Running a Simulation from the Command Line" in the Simulink documentation.

Learning More About Simulink

Here are a few more suggestions to help you get started with Simulink:

- Browse through the Simulink documentation to get complete exposure to all of the capabilities of Simulink.
- Open the Simulink library as described in “Starting Simulink” on page 2-3. Build a few simple models using blocks from the Simulink and DSP Blockset libraries.
- Open some of the models in the DSP Blockset Demos library by typing

  ```
  demo blocksets dsp
  ```

  Most of the advanced demos have blocks that you can double-click to get information about the algorithm or implementation. The Demos library also contains easy-to-understand models that demonstrate some of the blockset’s elementary math and statistics blocks. In each case, just select **Start** from the **Simulation** menu to run the simulation.

3

Working with Signals

The first part of this chapter will help you understand how signals are represented in Simulink. It covers a number of topics that are especially important in DSP simulations, such as sample rates and frame-based processing:

- "Signal Concepts" on page 3-3
- "Sample Rates and Frame Rates" on page 3-16

The second part of the chapter explains the practical aspects of how to create, construct, import, export, and view signals:

- "Creating Signals" on page 3-33
- "Constructing Signals" on page 3-42
- "Deconstructing Signals" on page 3-54
- "Importing Signals" on page 3-62
- "Exporting Signals" on page 3-72
- "Viewing Signals" on page 3-80

The last part of the chapter deals with the advanced topic of delay and latency:

- "Delay and Latency" on page 3-85

Signal Concepts

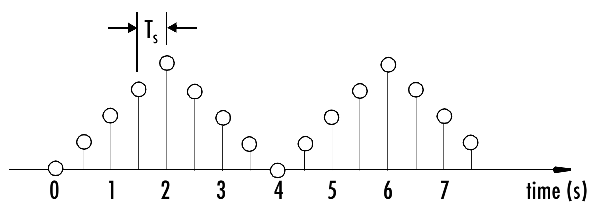
Simulink models can process both discrete-time and continuous-time signals, although models that are built with the DSP Blockset are often intended to process only discrete-time signals. The next few sections cover the following topics:

- "Discrete-Time Signals" — A brief introduction to some of the common terminology used for discrete-time signals, and a discussion of how discrete-time signals are represented within Simulink
- "Continuous-Time Signals" on page 3-9 — An explanation of how continuous-time signals are treated by various blocks in the DSP Blockset
- "Multichannel Signals" on page 3-11 — A description of how multichannel signals are represented in Simulink
- "Benefits of Frame-Based Processing" on page 3-14 — An explanation of how frame-based processing achieves higher throughput rates

Discrete-Time Signals

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.

For example, 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.



The following sections provide definitions for a number of terms commonly used to describe the time and frequency characteristics of discrete-time signals, and explain how these characteristics relate to Simulink models:

- "Time and Frequency Terminology"
- "Discrete-Time Signals in Simulink" on page 3-5

Time and Frequency Terminology

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 table below, are frequently used in Chapter 5, "DSP Block Reference," to describe the way that various blocks operate on sample-based and frame-based signals.

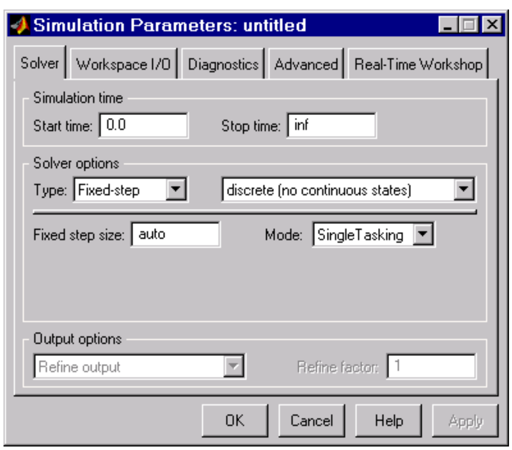
Term	Symbol	Units	Notes
Sample period	T_s T_{si} T_{so},	Seconds	The time interval between consecutive samples in a sequence, as the input to a block (T_{si}) or the output from a block (T_{so}).
Frame period	T_f T_{fi} T_{fo}	Seconds	The time interval between consecutive frames in a sequence, as the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

Term	Symbol	Units	Notes
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 dialog boxes, the term *sample time* is used to refer to the *sample period*, T_s. An example is the **Sample time** parameter in the Signal From Workspace block, which specifies the imported signal's sample period.

Discrete-Time Signals in Simulink

Simulink allows you to select from among several different simulation solver algorithms through the **Solver options** controls of the **Solver** panel in the **Simulation Parameters** dialog box. The selections that you make here determine how discrete-time signals are processed in Simulink.



The following sections explain the parameters available in this dialog box:

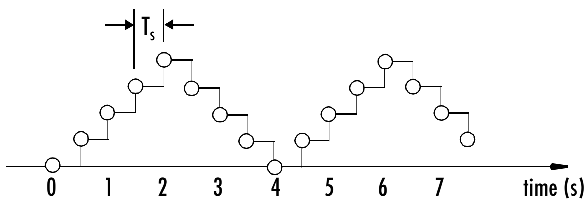
- "Recommended Settings for Discrete-Time Simulations"
- "Additional Settings for Discrete-Time Simulations" on page 3-7
- "Cross-Rate Operations in Variable-Step and Fixed-Step SingleTasking Modes" on page 3-7
- "Sample Time Offsets" on page 3-9

Recommended Settings for Discrete-Time Simulations. The recommended **Solver options** settings for DSP simulations are

- **Type** = **Fixed-step discrete**
- **Fixed step size** = `auto`
- **Mode** = **SingleTasking**

You can automatically set the above solver options for all new models by running the `dspstartup` M-file. See Appendix C, "Configuring Simulink for DSP Systems" for more information.

In **Fixed-step SingleTasking** mode, discrete-time signals *differ* from the prototype described in "Discrete-Time Signals" on page 3-3 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 in Variable-Step and Fixed-Step SingleTasking Modes" on page 3-7.

Additional Settings for Discrete-Time Simulations. It is worthwhile 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 most accurately model the prototypical discrete signal described in “Discrete-Time Signals” on page 3-3. In particular, when these settings are in effect, discrete signals 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.

To perform cross-rate operations like the addition of two signals with different sample rates, you must *explicitly* convert the two signals to a common sample rate. There are several blocks provided for precisely this purpose in the Signal Operations and Multirate Filters libraries. See “Converting Sample Rates and Frame Rates” on page 3-20 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 is selected, discrete time signals remain defined between sample times, just as in the **Fixed-step SingleTasking** setting previously described in “Recommended Settings for Discrete-Time Simulations” on page 3-6. In this mode, cross-rate operations are allowed by Simulink.

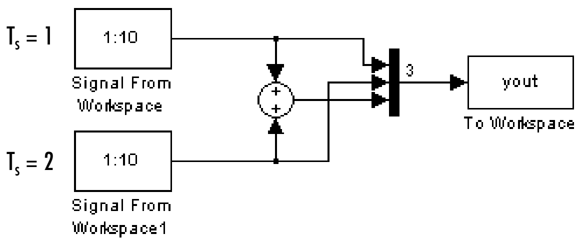
In the **Fixed-step Auto** setting, Simulink automatically selects a tasking mode (single-tasking or multitasking) that is best suited to the model. See “Simulink Tasking Mode” on page 3-91 for a description of the criteria that Simulink uses to make this decision. For the typical model containing multiple rates, Simulink selects the multitasking mode.

Cross-Rate Operations in Variable-Step and Fixed-Step SingleTasking Modes. In the Simulink **Variable step** and **Fixed-step SingleTasking** modes, a discrete-time signal is 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.

Note In the recommended `dspstartup` settings, **SingleTask rate transition** is set to **Error** in the **Diagnostics** pane in the **Simulation Parameters** dialog box. Thus, in the `dspstartup` configurations, cross-rate operations will generate errors even though the solver is in fixed-step single-tasking mode.

Example: Cross-Rate Operations. Consider the model below, which sums two signals having different sample periods. The fast signal (T_s=1) has sample times 1, 2, 3, ..., and the slow signal (T_s=2) has sample times 1, 3, 5, ...

This example will generate an error under the `dspstartup` settings, as explained in the previous Note.



The output, `yout`, is a matrix containing the fast signal (T_s=1) in the first column, the slow signal (T_s=2) in the second column, and the sum of the two in the third column:

```
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
```

As expected, the slow signal (second column) changes once every two seconds, half as often as the fast signal. Nevertheless, it has a defined value at every moment inbetween 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.

Sample Time Offsets. 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 DSP systems, and blocks from the DSP Blockset do not support them.

Continuous-Time Signals

Most signals in a DSP model are discrete-time signals, and all of the blocks in the DSP Blockset accept discrete-time inputs. However, many blocks can also operate on continuous-time signals, whose values vary continuously with time. Similarly, most blocks *generate* discrete-time signals, but some also generate continuous-time signals.

The sampling behavior of a particular block (continuous or discrete) determines which other blocks you can connect as an input or output. The following sections describe the behavior for two types of blocks:

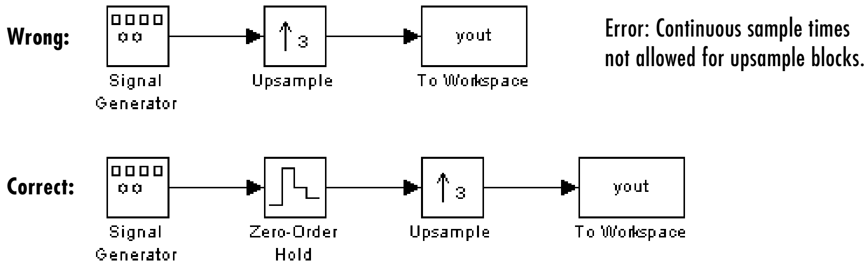
- "Source Blocks"
- "Nonsource Blocks" on page 3-10

See the online block reference pages for information about the particular sample characteristics of each block in the blockset.

Source Blocks

Source blocks are those blocks that generate or import signals in a model. Most source blocks appear in the DSP Sources library. See section "Importing Signals" on page 3-62 to fully explore the features of these blocks.

Continuous-Time Source Blocks. The sample period for continuous-time source blocks is set internally to zero, which indicates a continuous-time signal. The Signal Generator block in Simulink is an example of a continuous-time source block. Continuous-time signals are rendered in black when **Sample time colors** is selected from the **Format** menu. When connecting such blocks to discrete-time blocks, you may need to interpose a Zero-Order Hold block to discretize the signal (see the following diagram). Specify the desired sample period for the signal in the **Sample time** parameter of the Zero-Order Hold block.



The Triggered Signal From Workspace block is also a continuous-time block.

Discrete-Time Source Blocks. Discrete-time source blocks such as Signal From Workspace require a discrete (nonzero) sample period to be specified in the block's **Sample time** parameter. Simulink generates an error if a zero value is specified for the **Sample time** parameter of a discrete-time source block.

Nonsource Blocks

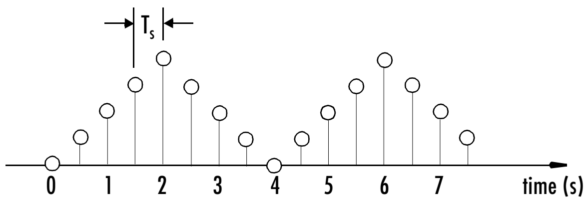
All nonsource blocks in the DSP Blockset accept discrete signals, and inherit the sample period of the input. Others additionally accept continuous-time discrete signals.

Discrete-Time Nonsource Blocks. Discrete-time nonsource blocks can accept only discrete-time inputs, and generate only discrete-time outputs. Examples are all of the resampling and delay blocks, including Upsample and Integer Delay. A discrete-time nonsource block *inherits* the sample period and sample rate of its driving block (the block supplying its input). For example, if the driving block's sample period is 0.5 second, the inheriting block also executes at 0.5 second intervals. Simulink generates an error if a continuous input is connected to a discrete-only block.

Continuous/Discrete Nonsource Blocks. In the continuous/discrete blocks, continuous-time inputs generate continuous-time outputs, and discrete-time inputs generate discrete-time outputs. Examples are the Complex Exponential and dB Gain blocks. The nonsource *triggered* blocks such as Triggered Delay Line are also in this category.

Multichannel Signals

The following figure shows the prototypical discrete-time signal discussed in "Discrete-Time Signals" on page 3-3. If this signal were propagated through a model sample-by-sample, rather than in batches of samples, it would be called *sample-based*. It would also be called *single-channel*, because there is only one independent sequence of numbers.



In practice, signal samples are frequently transmitted in batches, or frames, and several channels of data are often transmitted simultaneously. Hence, the general signal is frame-based and multichannel.

The following sections explain how sample-based and frame-based multichannel signals are represented in Simulink:

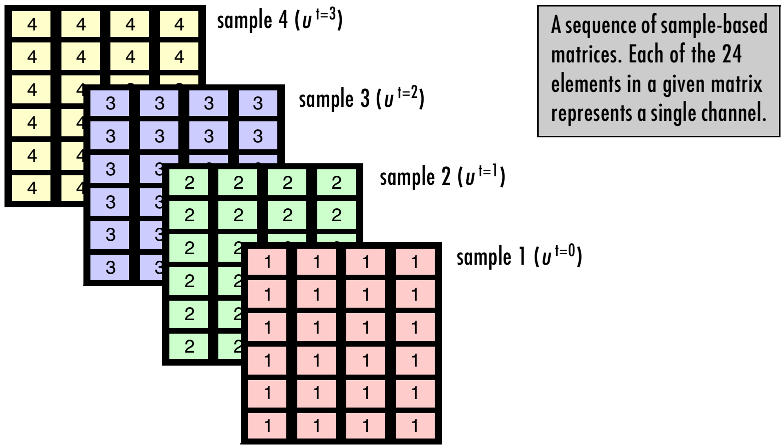
- "Sample-Based Multichannel Signals"
- "Frame-Based Multichannel Signals" on page 3-12

The representation of single-channel signals follows naturally as a special case (one channel) 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.



Then 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:

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

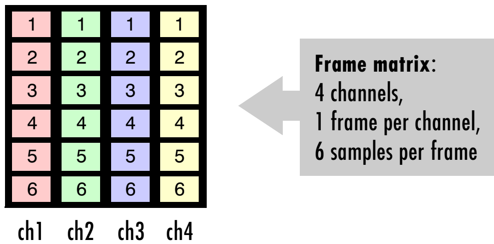
See the following sections for information about working with sample-based multichannel signals:

- “Creating Signals” on page 3-33
- “Constructing Signals” on page 3-42
- “Deconstructing Signals” on page 3-54
- “Importing Signals” on page 3-62
- “Exporting Signals” on page 3-72
- “Viewing Signals” on page 3-80

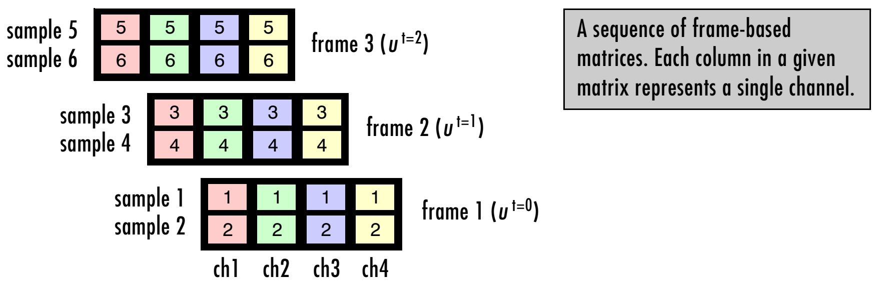
Frame-Based Multichannel Signals

Frame-based multichannel signals are also 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.

This is a simple structure, as illustrated below for a sample 6-by-4 frame matrix.



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.



Then 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:

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

See the following sections for information about working with frame-based multichannel signals:

- "Creating Signals" on page 3-33
- "Constructing Signals" on page 3-42
- "Deconstructing Signals" on page 3-54

- “Importing Signals” on page 3-62
- “Exporting Signals” on page 3-72
- “Viewing Signals” on page 3-80

Benefits of Frame-Based Processing

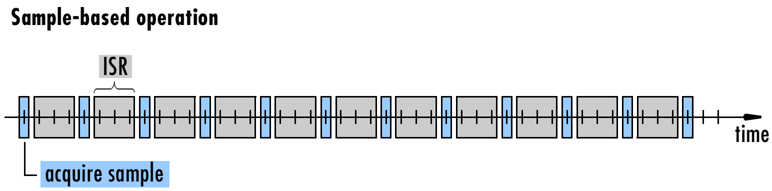
Frame-based processing is an established method of accelerating both real-time systems and simulations.

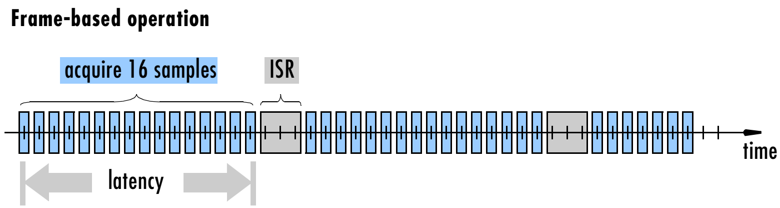
Accelerating Real-Time Systems

Framed-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 will introduce 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.

Accelerating Simulations

Simulation 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.

Sample Rates and Frame Rates

Sample rates are an important issue in most DSP models, especially in systems incorporating rate conversions. Fortunately, in most cases, when you build a Simulink model you only need to worry about setting sample rates in the source blocks, such as Signal From Workspace; Simulink automatically computes the appropriate sample rates for all downstream blocks.

Nevertheless, it is important to become familiar with the concepts of “sample rate” and “frame rate” as they apply in the Simulink world. The next sections cover the following important topics:

- “Sample Rate and Frame Rate Concepts”
- “Inspecting Sample Rates and Frame Rates” on page 3-17
- “Converting Sample Rates and Frame Rates” on page 3-20
- “Changing Frame Status” on page 3-31

Sample Rate and Frame Rate Concepts

The *input frame period* (T_{fi}) of a frame-based signal is the time interval between consecutive vector or matrix inputs to a block. This interval is what the Probe block displays when you connect it to a frame-based input line. 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. This interval is what the Probe block displays when you connect it to a frame-based output line. (See “Inspecting Sample Rates and Frame Rates” on page 3-17 for more about using the Probe block.)

In contrast, the sample period, T_s, is the time interval between individual samples in a frame, which is necessarily 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:

$$T_s = T_f / M$$

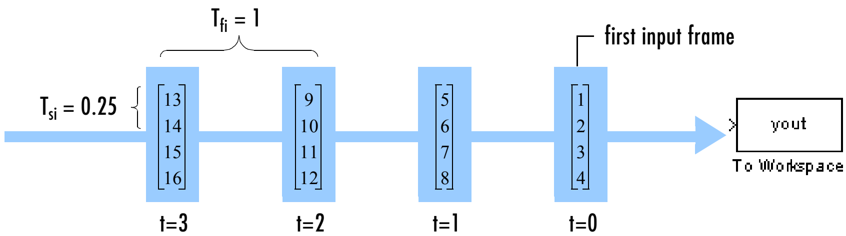
More specifically, the sample periods of inputs (T_{si}) and outputs (T_{so}) are related to their respective frame periods by

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

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

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

The illustration below shows a one-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. A Probe block connected to this signal would display the frame period T_{fi} = 1.



In most cases, the sequence sample period T_{si} is of primary interest, 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.

For information on converting a signal from one sample rate or frame rate to another, see "Converting Sample Rates and Frame Rates" on page 3-20.

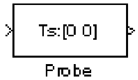
Inspecting Sample Rates and Frame Rates

When constructing a frame-based or multirate model, it is often helpful to check the rates that Simulink computes for different signals. There are two basic ways to inspect the sample rates and frame rates in a model. These are described in the following sections:

- "Using the Probe Block to Inspect Rates"
- "Using Sample Time Color Coding to Inspect Sample Rates" on page 3-19

Using the Probe Block to Inspect Rates

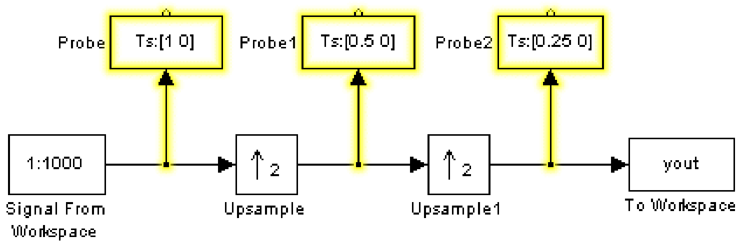
Connect the Simulink Probe block to any line to display the period of the signal on that line. The period is displayed in the block icon itself (together with the line width and data type, if desired), making it easy to verify that the sample rates in the model are what you expect them to be. When the line width and data type displays are suppressed (by clearing the appropriate check boxes in the block dialog box), the Probe block looks like this.



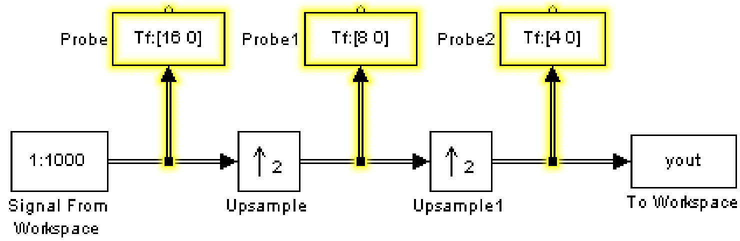
The block displays the label `Ts` or `Tf`, followed by a two-element vector. The first (left) element is the period of the signal being measured. The second (right) is the signal's sample time offset, which is usually `0`, as explained in "Sample Time Offsets" on page 3-9.

For sample-based signals, the value shown in the Probe block icon is the sample period of the sequence, T_s. For frame-based signals, the value shown in the Probe block icon is the frame period, T_f. The difference between sample rates and frame rates is explained in "Sample Rate and Frame Rate Concepts" on page 3-16.

Probe Block Example: Sample-Based. The three Probe blocks in the sample-based model below verify that the signal's sample period is halved with each upsample operation: The output from the Signal From Workspace block has 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.



Probe Block Example: Frame-Based. The three Probe blocks in the frame-based model below again verify that the signal's sample period is halved with each upsample operation: The output from the Signal From Workspace block has a frame period of 16 seconds, the output from the first Upsample block has a frame period of 8 seconds, and the output from the second Upsample block has a sample period of 4 seconds.

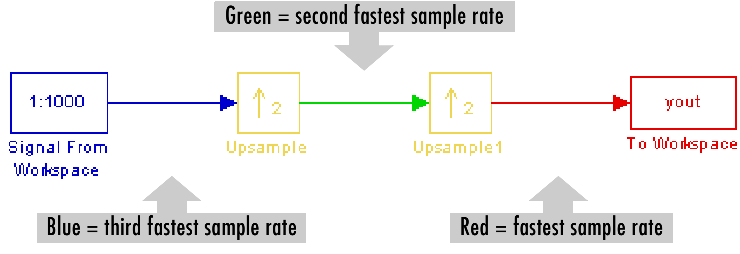


Note that the sample rate conversion is implemented through a change in the frame period rather than the frame size. This is because the **Frame-based mode** parameter in the Upsample blocks is set to **Maintain input frame size** rather than **Maintain input frame rate**. See "Converting Sample Rates and Frame Rates" on page 3-20 for more information.

Using Sample Time Color Coding to Inspect Sample Rates

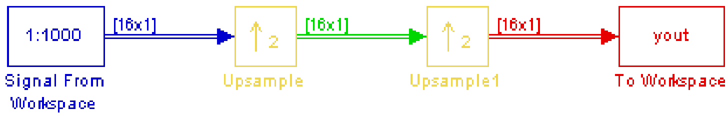
Turn on the sample time color coding option in Simulink by selecting **Sample time colors** from the **Format** menu. For sample-based signals, this assigns each sample rate a different color. For frame-based signals, this assigns each frame rate a different color.

Sample Time Color Coding Example: Sample-Based. Here is the sample-based model from "Probe Block Example: Sample-Based" on page 3-18 with the Probe blocks removed and sample time color coding turned on.



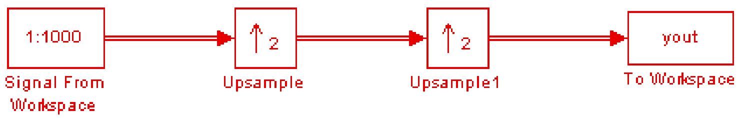
Since every sample-based signal in this model has a different sample rate, each signal is assigned a different color.

Sample Time Color Coding Example: Frame-Based. Here's the frame-based model from "Probe Block Example: Frame-Based" on page 3-18 with the Probe blocks removed and sample time color coding turned on.



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.

If the Upsample blocks are instead set to **Maintain input frame rate**, then every signal in the model shares the same frame rate, and as a result, every signal is coded with the same color.



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

Converting Sample Rates and Frame Rates

In a DSP Blockset model, there are two types of periods that you will commonly be concerned with: sample periods and frame periods. The input and output sample periods of a block (T_{si} and T_{so}, respectively) are related to the input and output frame periods (T_{fi} and T_{fo}, respectively) 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 buffering and rate-conversion capabilities of the DSP Blockset generally allow you to independently vary any two of the three parameters (T_{so}, T_{fo}, M_o). In most cases, the sample period and the frame size are the two parameters of primary interest; the frame period is simply a consequence of your choices for the other two.

There are two common types of operations that impact the frame and sample rates of a signal:

- Direct rate conversions

  Direct rate conversions, such as upsampling and downsampling, are a feature of most DSP systems, and can be implemented by altering either the frame rate or the frame size of a signal.
- Frame rebuffering

  The principal purpose of frame rebuffering is to alter the frame size of a signal, usually to improve simulation throughput. By redistributing the signal samples to frames of a new size, rebuffering usually changes either the sample rate or frame rate of the signal.

Both operations are discussed in the following sections, along with ways to avoid *unintentional* rate conversions:

- "Direct Rate Conversion"
- "Frame Rebuffering" on page 3-24
- "Avoiding Unintended Rate Conversions" on page 3-28

You may also want to look at the Sample Rate Conversion demo, `dspsrcnv.mdl`.

Note Technically, when a Simulink model contains signals with various frame rates, the model is called *multirate*. You can find a discussion of multirate models in "Delay and Latency" on page 3-85 and in the topic on discrete time systems in the Simulink documentation.

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

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

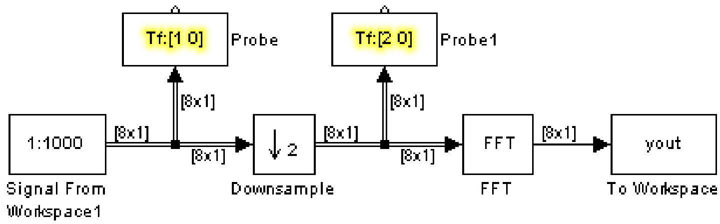
Rate Conversion Blocks. The following table lists the principal rate conversion blocks in the DSP Blockset. Blocks marked with an asterisk (*) offer the option of changing the rate by either adjusting the frame size or frame rate.

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
Wavelet Analysis	Filtering / Multirate Filters
Wavelet Synthesis	Filtering / Multirate Filters

The following examples illustrate the two sample rate conversion modes:

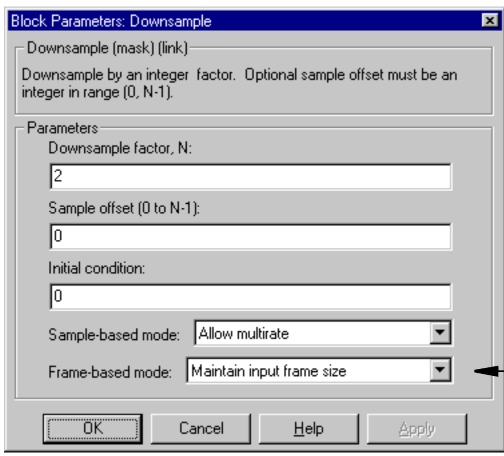
- "Example: Rate Conversion by Frame-Rate Adjustment"
- "Example: Rate Conversion by Frame-Size Adjustment" on page 3-23

Example: Rate Conversion by Frame-Rate Adjustment. A common example of direct rate conversion is shown in the model below, where the signal is directly downsampled to half its original rate by a Downsample block. The values next to input and output ports are the signal dimensions, displayed by selecting **Signal dimensions** from the model window's **Format** menu.



The sample period and frame size of the original signal are set to 0.125 second and 8 samples per frame, respectively, by the **Sample time** and **Samples per frame** parameters in the Signal From Workspace block. This results in a frame rate of 1 second (0.125∗8), as shown by the first Probe block.

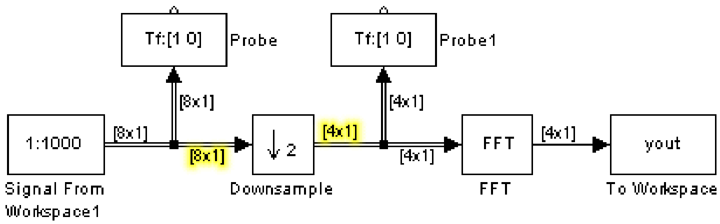
The Downsample block is configured to downsample the signal by changing the frame *rate* rather than the frame *size*. The dialog box with this setting is shown below.



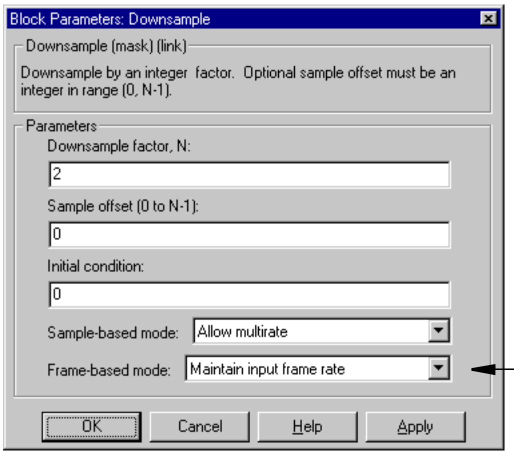
Maintain input frame size: Downsample the signal by changing the frame *rate*.

The second Probe block in the model verifies that the output from the Downsample block has a frame period of 2, twice that of the input (i.e., half the rate). As a result, the sequence sample period is doubled to 0.25 second without any change to the frame size.

Example: Rate Conversion by Frame-Size Adjustment. The model from "Example: Rate Conversion by Frame-Rate Adjustment" on page 3-22 is shown again below, but this time with the rate conversion implemented by adjusting the frame size, rather than the frame rate.



As before, the frame rate of the original signal is 1 second (0.125∗8), shown by the first Probe block. Now the Downsample block is configured to downsample the signal by changing the frame *size* rather than the frame *rate*. The dialog box with this setting is shown below.



Maintain input frame rate: Downsample the signal by changing the frame *size*.

The line width display on the Downsample output port verifies that the downsampled output has a frame size of 4, half that of the input. As a result, the sequence sample period is doubled to 0.25 second without any change to the frame rate.

Frame Rebuffering

Buffering operations provide another mechanism for rate changes in DSP models. The purpose of many buffering operations is to adjust the frame size of the signal, M, without altering the sequence sample rate T_s. This usually results in a change to the signal's frame rate, T_f, according to the relation

$$T_f = MT_s$$

However, this is only true when the *original signal is preserved* in the buffering operation, with no samples added or deleted. Buffering operations that generate overlapping frames, or that only partially unbuffer frames, alter the data sequence by adding or deleting samples. In such cases, the above relation is not valid.

Buffering Blocks. The following table lists the principal buffering blocks in the DSP 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

The following sections discuss two general classes of buffering operations:

- "Buffering with Preservation of the Signal"
- "Buffering with Alteration of the Signal" on page 3-26

Buffering with Preservation of the Signal. There are various reasons that you may 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 DSP algorithm in the model. In this case, you would want to rebuffer the signal to a frame size more appropriate for the intended operations, but without introducing any change to the data or sample rate.

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 **Buffer size** parameter.

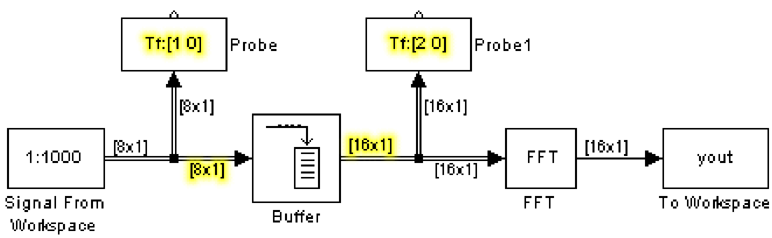
The Unbuffer block is specialized for completely unbuffering a frame-based signal to its sample-based equivalent, and always preserves the signal's data and sample period:

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

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 ($T_{so} = T_{si}$).

Example: Buffering with Preservation of the Signal. In the model below, 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 doubles the frame period from 1 to 2 seconds, but does not change the sample period of the signal ($T_{so} = T_{si} = 0.125$).



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. There are many instances when this type of buffering is desirable, for example when creating sliding windows by overlapping consecutive frames of a signal, or selecting 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.

- Buffer 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 **Buffer size** parameter. As a result, the new output sample period is

  $$T_{so} = \frac{(M_o - L)T_{si}}{M_o}$$

- Delay Line 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

  $$T_{so} = \frac{T_{si}}{M_o}$$

- Variable Selector 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

  $$T_{so} = \frac{M_i T_{si}}{M_o}$$

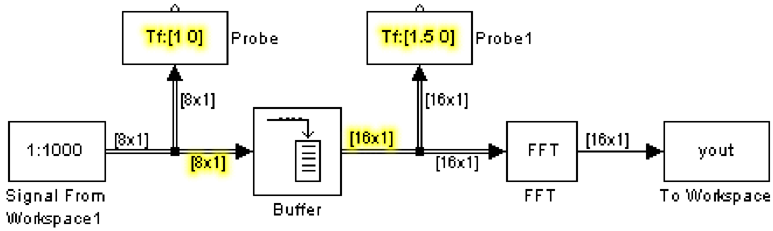
  where M_o is the length of the block's output, determined by the **Elements** vector.
- Zero Pad adds samples to the sequence by appending zeros to each frame when **Zero pad along** is set to **Columns**. The output and input frame periods are the same, $T_{fo} = T_{fi}$, and the new output sample period 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.

Example: Buffering with Alteration of the Signal. In the model below, 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 an overlap of 4.



The relation for the output frame period for the Buffer block is

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

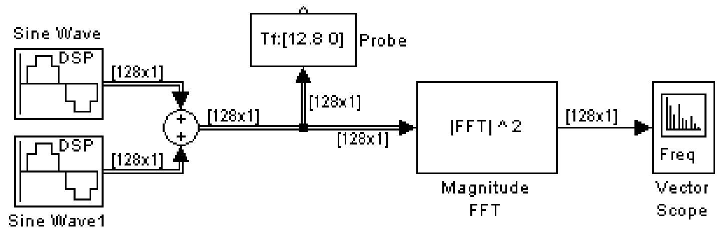
which indicates that T_{fo} should be (16-4)*0.125, or 1.5 seconds, as confirmed by the second Probe block. The sample period of the signal at the output of the Buffer block is no longer 0.125 second, but rather 0.0938 second (i.e., 1.5/16). Thus, both the signal's data and the signal's sample period have been altered by the buffering operation.

Avoiding Unintended Rate Conversions

The previous sections discussed a number of the blocks that are responsible for rate conversions. It is important to be aware of where in a model these rate conversions are taking place; in a few cases, *unintentional* rate conversions can produce misleading results. The following pair of examples illustrate how unintended rate conversion can occur:

- "Example 1: No Rate Conversion"
- "Example 2: Unintended Rate Conversion" on page 3-30

Example 1: No Rate Conversion. The model below plots the magnitude FFT of a signal composed of two sine waves, with frequencies of 1 Hz and 2 Hz.



To build the model, configure one Sine Wave block with **Frequency** = 1, and the other with **Frequency** = 2. In addition, both Sine Wave blocks should have the following settings:

- **Sample time** = 0.1
- **Samples per frame** = 128

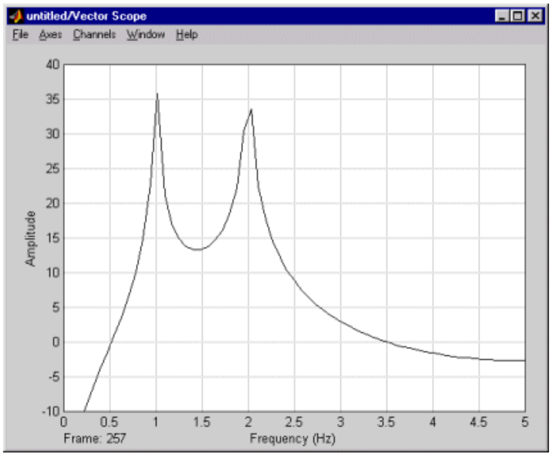
The frame period of the resulting summed sinusoid is 12.8 seconds (i.e., 128*0.1), which is confirmed by the Probe block when the model is updated.

Select **Inherit FFT length from input dimensions** in the Magnitude FFT block. This setting instructs the block to use the input frame size (128) as the FFT length (which is also the output size).

Configure the Vector Scope block as follows:

- Select **Frequency** from the **Input domain** parameter.
- Select the **Axis properties** check box to expose the **Axis properties** panel.
- Set **Minimum Y-limit** to -10.
- Set **Maximum Y-limit** to 40.

The plot generated by the Vector Scope block is shown below.

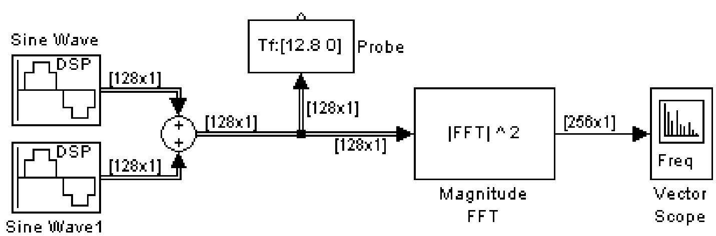


The Vector Scope block uses the input frame size (128) and period (12.8) to deduce the original signal's sample period (0.1), which allows it to *correctly* display the peaks at 1 Hz and 2 Hz.

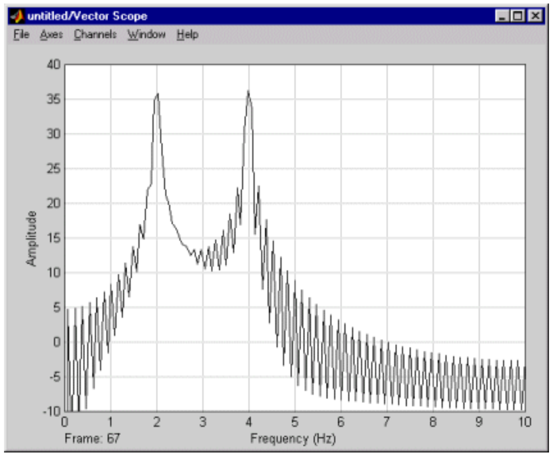
Example 2: Unintended Rate Conversion. Now alter the previous example by setting the Magnitude FFT block parameters as follows:

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

This setting instructs the block to zero-pad the length-128 input frame to a length of 256 before performing the FFT. The signal dimension display on the new version of the model shows that the output of the Magnitude FFT block is now a length-256 frame.



The plot generated by the Vector Scope block is shown below.



In this case, based on the input frame size (256) and period (12.8), the Vector Scope block calculates the original signal's sample period to be 0.05 second (12.8/256), which is *wrong*. As a result, the spectral peaks appear at the incorrect frequencies, 2 Hz and 4 Hz rather than 1 Hz and 2 Hz.

The problem is that the zero-pad operation performed by the Magnitude FFT block halves the sample period of the sequence by appending 128 zeros to each frame. The Vector Scope block, however, needs to know the sample period of the *original* signal. The problem is easily solved by changing the **Sample time of original time series** setting in the **Axis properties** panel of the Vector Scope block to the actual sample period of `0.1`. The plot generated with this setting is identical to the first Vector Scope plot above.

In general, be aware that when you do zero-padding or overlapping buffering you are changing the sample period of the signal. As long as you keep this in mind, you should be able to anticipate and correct problems like the one above.

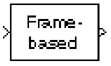
Changing Frame Status

The *frame status* of a signal refers to whether the signal is sample-based or frame-based. In a Simulink model, the frame status is symbolized by a single line, $\rightarrow$, for a sample-based signal and a double line, $\Rightarrow$, for a frame-based signal.

In most cases, the appropriate way to convert a sample-based signal to a frame-based signal is by using the Buffer block, and the appropriate way to convert a frame-based signal to a sample-based signal is by using the Unbuffer block. See the following sections for more information about these methods:

- “Buffering Sample-Based and Frame-Based Signals” on page 3-47
- “Unbuffering a Frame-Based Signal into a Sample-Based Signal” on page 3-60

On occasion it may be desirable to change the frame status of a signal without performing a buffering operation. You can do this by using the Frame Status Conversion block in the Signal Attributes library.



The **Output signal** parameter (or the signal at the optional `Ref` input port) determines the frame status of the output If the frame status of the input differs from the **Output signal** setting, then the frame status is altered as specified. If the frame status of the input is the same as that specified by the **Output signal** parameter, then no change is made to the signal.

The block’s input and output port rates are the same, and because the block does not make any sample rate accommodation, the sample rate of the signal is generally not preserved under a change of frame status. (The exception to this rule occurs when a sample-based signal is converted to a frame-based signal with frame size 1, or vice versa.)

See the Frame Status Conversion block’s reference page for complete information.

Creating Signals

There are a variety of different ways to create signals using Simulink and DSP blocks. The following sections explore the most common techniques:

- "Creating Signals Using Constant Blocks" on page 3-33
- "Creating Signals Using Signal Generator Blocks" on page 3-36
- "Creating Signals Using the Signal From Workspace Block" on page 3-38

The above sections discuss creating signals (single-channel and multichannel) using source blocks. For information about constructing multichannel signals from existing single-channel signals, see the following sections:

- "Constructing Multichannel Sample-Based Signals" on page 3-42
- "Constructing Multichannel Frame-Based Signals" on page 3-45

Creating Signals Using Constant Blocks

A *constant* signal is a sample-based signal in which successive samples are identical, or a frame-based signal in which successive frames are identical. The DSP Sources library provides the following blocks for creating sample-based and frame-based constant signals:

- Constant Diagonal Matrix
- Constant Ramp
- DSP Constant
- Identity Matrix
- Window Function

Although some of these blocks generate continuous-time outputs and some generate discrete-time outputs, in each case the output of the block remains constant throughout the simulation.

The most versatile of these blocks is the DSP Constant, which is discussed further in the following example. See Chapter 5, "DSP Block Reference," for complete explanation of all the constant blocks.

For information about creating signals with other types of blocks, see the following sections:

- "Creating Signals Using Signal Generator Blocks" on page 3-36

- “Creating Signals Using the Signal From Workspace Block” on page 3-38

For information about importing signals, see the following sections:

- “Importing a Multichannel Sample-Based Signal” on page 3-62
- “Importing a Multichannel Frame-Based Signal” on page 3-68

Example: Creating Signals with the DSP Constant Block

The DSP Constant block has the following parameters:

- **Constant value**
- **Interpret vector parameters as 1-D**
- **Sample mode**
- **Sample time**
- **Frame-based output**

To generate a constant matrix signal, simply enter the desired matrix in the **Constant value** parameter using standard MATLAB notation. Some common examples of the MATLAB matrix notation are shown below:

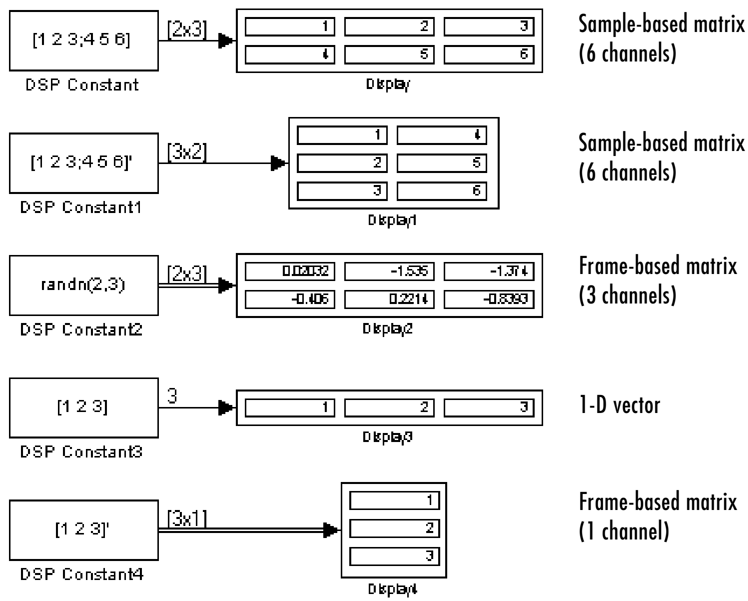
```
[1 2 3;4 5 6]       % A 2-by-3 matrix

[1 2 3;4 5 6]'      % The transpose, a 3-by-2 matrix

randn(2,3)          % A 2-by-3 matrix with random elements

[1 2 3]             % A 1-by-3 row vector

[1 2 3]'            % The transpose, a 3-by-1 column vector
```

As with all numerical parameters, you can also enter any valid MATLAB variable or expression that evaluates to a matrix. See the MATLAB documentation for a thorough introduction to constructing and indexing matrices.

The **Interpret vector parameters as 1-D** and **Frame-based output** parameters are discussed following the example below. See the DSP Constant block’s reference page for information about the **Sample mode** and **Sample time** parameters.

The model below shows five DSP Constant blocks, each generating one of the constant signals listed above. Two of the blocks have nondefault settings for the other parameters: The third block (DSP Constant2) has the **Frame-based**

output check box selected, and the fourth block (DSP Constant3) has the **Interpret vector parameters as 1-D** check box selected.



In addition to the various output dimensions in the model, you can observe three different kinds of signals:

- *Sample-based matrix signal* — The DSP Constant and DSP Constant1 blocks generate sample-based matrices (2-by-3 and 3-by-2, respectively) because the **Frame-based output** check box in those blocks is *not* selected. The sample-based matrices can each be considered to each have six independent channels.
- *Frame-based matrix signal* — The DSP Constant2 and DSP Constant4 blocks generate frame-based matrices (2-by-3 and 3-by-1, respectively, and represented by double lines) because the **Frame-based output** check box in those blocks is selected. The 2-by-3 frame-based matrix is considered to have three independent channels, each containing two consecutive samples. The 3-by-1 frame-based matrix (column vector) is considered to have one independent channel, containing three consecutive samples.
- *1-D vector signal* — The DSP Constant3 block generates a length-3 1-D vector signal because the **Interpret vector parameters as 1-D** check box in

that block is selected. This means that the output *is not a matrix*. However, most nonsource DSP 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. The **Interpret vector parameters as 1-D** parameter is ignored when **Frame-based output** is selected, or when a matrix is specified for the **Constant value** parameter.

See “Multichannel Signals” on page 3-11 for more information about the representation of sample-based and frame-based data.

Creating Signals Using Signal Generator Blocks

The DSP Sources library provides the following blocks for automatically generating common sample-based and frame-based signals:

- Chirp
- Counter
- Discrete Impulse
- Multiphase Clock
- N-Sample Enable
- Sine Wave

One of the most commonly used of these is the Sine Wave block, which is discussed further in the example below. See Chapter 5, “DSP Block Reference,” for a complete explanation of the other signal generation blocks. The Simulink Sources library offers a collection of continuous-time signal generation blocks that you may also find useful. Consult the Simulink documentation for more information.

For more information about creating signals, see the following sections:

- “Creating Signals Using Constant Blocks” on page 3-33
- “Creating Signals Using the Signal From Workspace Block” on page 3-38

Example: Creating Signals with the Sine Wave Block

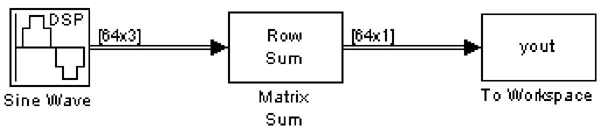
The Sine Wave block dialog box contains the following key parameters.

- **Amplitude**
- **Frequency**
- **Phase offset**
- **Sample time**
- **Samples per frame**

In the model below, a Sine Wave block generates a frame-based (multichannel) matrix containing three independent signals:

- Sine wave of amplitude 1 and frequency 100 Hz
- Sine wave of amplitude 3 and frequency 250 Hz
- Sine wave of amplitude 2 and frequency 500 Hz

Each channel has a frame size of 64 samples. The three signals are summed point-by-point by a Matrix Sum block, and exported to the workspace.



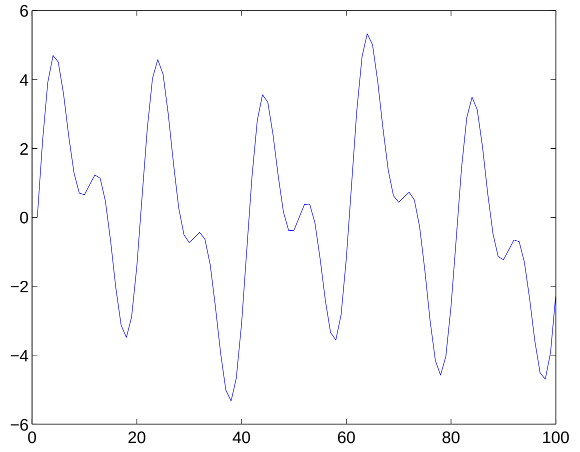
To build the model, set the **Sum along** parameter of the Matrix Sum block to **Rows**, and make the following parameter settings in the Sine Wave block:

- Set **Amplitude** to `[1 3 2]`. This specifies the amplitudes for three independent sinusoids (and therefore dictates a three-column output).
- Set **Frequency** to `[100 250 500]`. This specifies the frequency for each of the output sinusoids.
- Set **Sample time** to `1/5000`. (This is ten times the highest sinusoid frequency, and so satisfies the Nyquist criterion.)
- Set **Samples per frame** to `64`. This specifies a frame size of 64 for all sinusoids (and therefore dictates a 64-row output).

After running the model, you can look at a portion of the resulting summed sinusoid by typing

```
plot(yout(1:100))
```

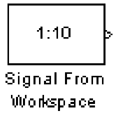
at the command line.



See "Multichannel Signals" on page 3-11 for more information about the representation of sample-based and frame-based data.

Creating Signals Using the Signal From Workspace Block

You can easily create custom signals using the Signal From Workspace block.



This block allows you to generate arbitrary sample-based and frame-based signals, as illustrated in the following examples:

- "Example 1: Generating Sample-Based Output" on page 3-39
- "Example 2: Generating Frame-Based Output" on page 3-40

As the name implies, the Signal From Workspace block is more commonly used to import custom signals from the workspace. See the following sections for more information:

- "Importing a Multichannel Sample-Based Signal" on page 3-62
- "Importing a Multichannel Frame-Based Signal" on page 3-68

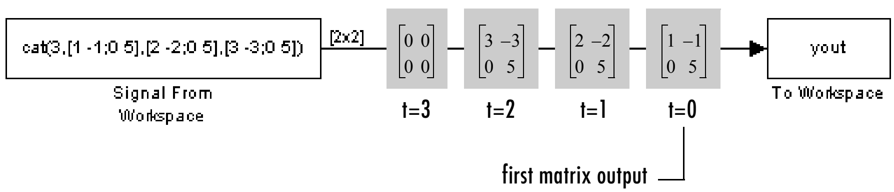
For more information about creating signals, see the following sections:

- “Creating Signals Using Constant Blocks” on page 3-33
- “Creating Signals Using Signal Generator Blocks” on page 3-36

Example 1: Generating Sample-Based Output

In the model below, the Signal From Workspace creates a four-channel sample-based signal with the following data:

- **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,...



To create the model, specify the following parameter values in the Signal From Workspace block:

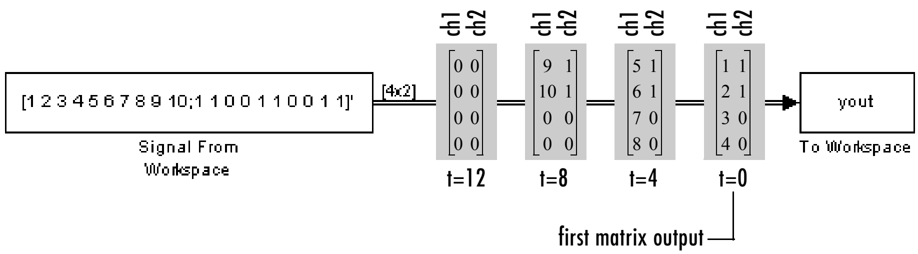
- **Signal** = `cat(3,[1 -1;0 5],[2 -2;0 5],[3 -3;0 5])`
- **Sample time** = `1`
- **Samples per frame** = `1`
- **Form output after final data value** = **Setting to zero**

The **Sample time** setting of `1` yields a sample-based output with sample period of 1 second. Each of the four elements in the matrix signal represents an independent channel (the channel numbering is arbitrary). The **Form output after final data value** parameter setting specifies that all outputs after the third are zero.

Example 2: Generating Frame-Based Output

In the model below, the Signal From Workspace creates a two-channel frame-based signal with the following data:

- **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,...



To create the model, specify the following parameter values in the Signal From Workspace block:

- **Signal** = `[1 2 3 4 5 6 7 8 9 10;1 1 0 0 1 1 0 0 1 1]'`
- **Sample time** = `1`
- **Samples per frame** = `4`
- **Form output after final data value** = **Setting to zero**

The **Sample time** setting of `1` and the **Samples per frame** setting of `4` yield a frame-based output with a frame size of 4 samples and a frame period of 4 seconds. The **Form output after final data value** parameter setting specifies that all outputs after the third frame are zero.

Note that the output of the To Workspace block, `yout`, is the original signal with appended zeros in each channel:

```
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
```

Constructing Signals

When you want to perform a given sequence of operations on several independent signals, it is frequently very convenient to group those signals together as a *multichannel signal*. Most DSP blocks accept multichannel signals, and process each channel independently. By taking advantage of this capability, you can do the same job with fewer blocks and have a cleaner, leaner model.

For example, if you need to filter each of four independent signals using a direct-form II transpose filter with the same coefficients, combine the signals into a multichannel signal, and run that multichannel signal into a Direct-Form II Transpose Filter block. The block will apply the filter to each channel independently.

The following sections explain how to construct multichannel signals from existing independent signals:

- "Constructing Multichannel Sample-Based Signals" on page 3-42
- "Constructing Multichannel Frame-Based Signals" on page 3-45

For information about creating multichannel signals using source blocks, see the following sections:

- "Creating Signals Using Constant Blocks" on page 3-33
- "Creating Signals Using Signal Generator Blocks" on page 3-36
- "Creating Signals Using the Signal From Workspace Block" on page 3-38

Constructing Multichannel Sample-Based Signals

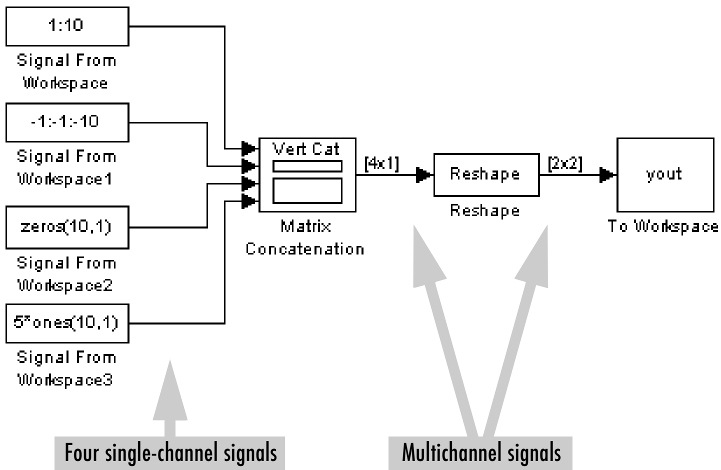
A sample-based signal with M∗N channels is represented by a sequence of M-by-N matrices. (The special case of M = N = 1 represents a single-channel signal.) Multiple individual signals can be combined into a multichannel matrix signal using the Matrix Concatenation block. Individual signals can be added to an existing multichannel signal in the same way. The following sections explain how to do this:

- "Constructing Sample-Based Multichannel Signals from Independent Sample-Based Signals" on page 3-43
- "Constructing Sample-Based Multichannel Signals from Existing Sample-Based Multichannel Signals" on page 3-44

Constructing Sample-Based Multichannel Signals from Independent Sample-Based Signals

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

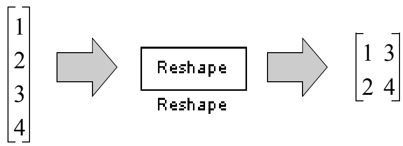
Example: Concatenating Single-Channel Signals. In the model below, four independent sample-based signals are combined into a 2-by-2 multichannel matrix signal.



To build the model, make the following parameter settings:

- In Signal From Workspace, set **Signal** = `1:10`
- In Signal From Workspace1, set **Signal** = `-1:-1:-10`
- In Signal From Workspace2, set **Signal** = `zeros(10,1)`
- In Signal From Workspace3, set **Signal** = `5*ones(10,1)`
- In Matrix Concatenation, set:
  - **Number of inputs** = `4`
  - **Concatenation method** = **Vertical**
- In Reshape, set:
  - **Output dimensionality** = **Customize**
  - **Output dimensions** = `[2,2]`

Each 4-by-1 output from the Matrix Concatenation block contains one sample from each of the four input signals. All four samples in the output correspond to the same instant in time. The Reshape block simply rearranges the samples into a 2-by-2 matrix. Note that the Reshape block works columnwise, so that a column vector input is reshaped as shown below.

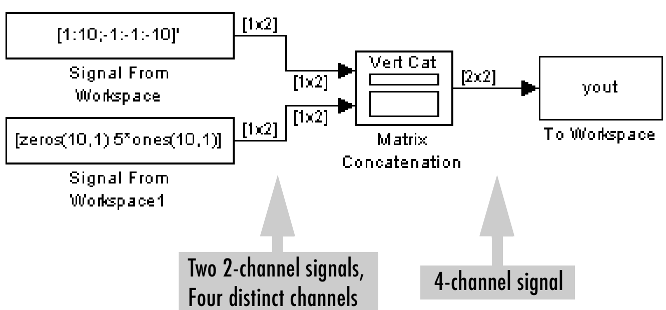


The 4-by-1 matrix and the 2-by-2 matrix in the above model represent the same sample-based four-channel signal. In some cases one representation may be more useful than the other. See "Sample-Based Multichannel Signals" on page 3-11 for more about sample-based signals.

Constructing Sample-Based Multichannel Signals from Existing Sample-Based Multichannel Signals

You can combine existing multichannel sample-based signals into a larger multichannel signal by using the Matrix Concatenation block in the Simulink Sources library.

Example: Concatenating Multichannel Signals. The model below shows two two-channel sample-based signals (four channels total) being combined into a 2-by-2 multichannel matrix signal.



To build the model, make the following parameter settings:

- In Signal From Workspace, set **Signal** = `[1:10;-1:-1:-10]'`

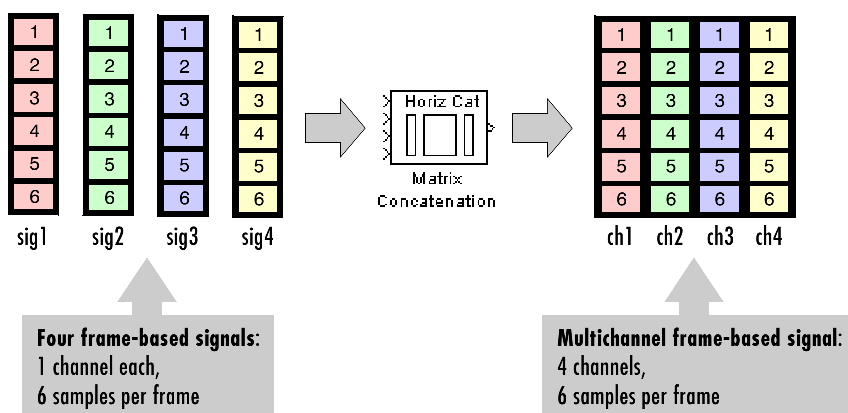
- In Signal From Workspace1, set **Signal** = `[zeros(10,1) 5*ones(10,1)]`
- In Matrix Concatenation, set:
  - **Number of inputs** = 2
  - **Concatenation method** = **Vertical**

Each 2-by-2 output from the Matrix Concatenation block contains both samples from each of the two input signals, so that all four samples in the output correspond to the same instant in time. See "Sample-Based Multichannel Signals" on page 3-11 for more about sample-based signals.

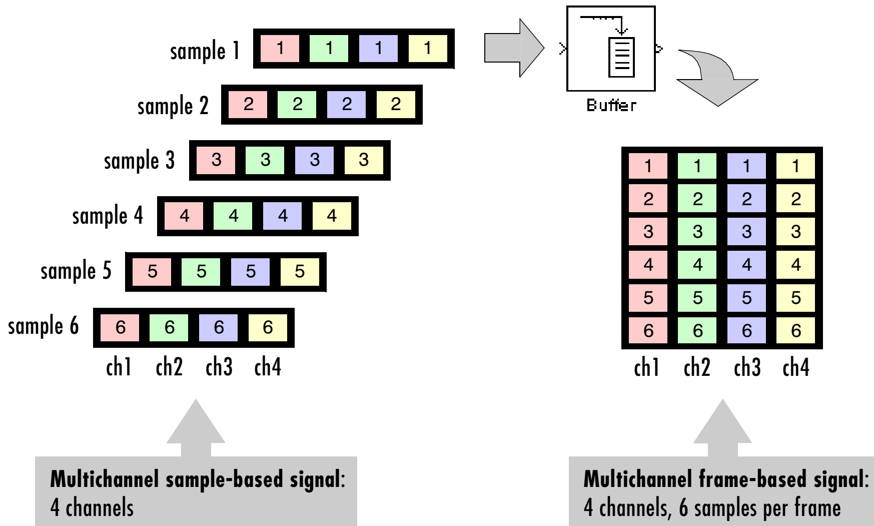
Constructing Multichannel Frame-Based Signals

A frame-based signal with N channels and frame size M is represented by a sequence of M-by-N matrices. (The special case of N = 1 represents a single-channel signal.) There are two basic ways to construct a multichannel frame-based signal from existing signals:

- *By horizontally concatenating existing frame-based signals* — 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.



- *By buffering existing sample-based or frame-based signals* — Multichannel sample-based and frame-based signals can be buffered into multichannel frame-based signals using the Buffer block in the Buffers library (in Signal Management).



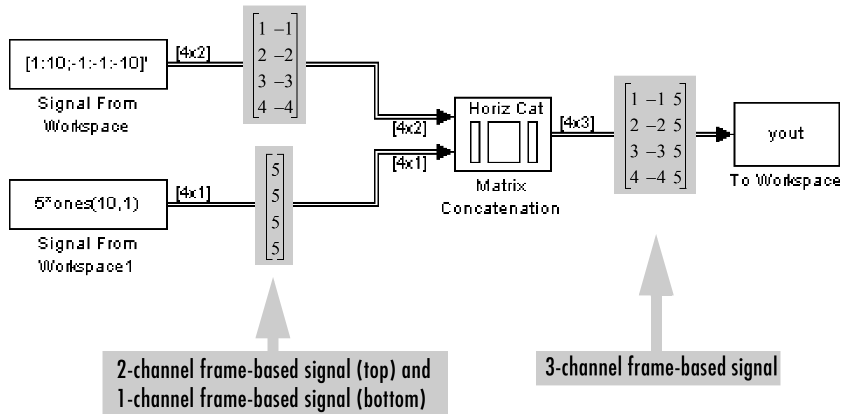
The following sections explain the two methods of constructing multichannel frame-based signals:

- "Concatenating Independent Frame-Based Signals into Multichannel Signals" on page 3-46
- "Buffering Sample-Based and Frame-Based Signals" on page 3-47

Concatenating Independent Frame-Based Signals into Multichannel Signals

You can combine existing frame-based signals into a larger multichannel signal by using the Matrix Concatenation block in the Simulink Sources library. All signals must have the same frame rate and frame size.

Example: Concatenating Frame-Based Signals. In the model below, a single-channel frame-based signal is combined with a two-channel frame-based signal to produce a three-channel frame-based signal.



To build the model, make the following parameter settings:

- In Signal From Workspace, set **Signal** = `[1:10;-1:-1:-10]'`
- In Signal From Workspace1, set **Signal** = `5*ones(10,1)`
- In Matrix Concatenation, set:
  - **Number of inputs** = 2
  - **Concatenation method** = **Horizontal**

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. See “Frame-Based Multichannel Signals” on page 3-12 for more about frame-based signals.

Note that you could also create or import the three-channel signal using just one Signal From Workspace block. See the following sections for more information:

- “Creating Signals Using the Signal From Workspace Block” on page 3-38
- “Importing a Multichannel Frame-Based Signal” on page 3-68

Buffering Sample-Based and Frame-Based Signals

You can buffer a multichannel sample-based or frame-based signal into a multichannel frame-based signal by using the Buffer block in the Buffers library (in Signal Management). The Buffer block has the following key parameters:

- **Output buffer size (per channel)**, M_o
- **Buffer overlap**, L
- **Initial conditions**

Buffering an N-channel (1-by-N or N-by-1) sample-based signal produces a M_o-by-N frame-based signal. Buffering an M_i-by-N frame-based signal (N channels and M_i samples per frame) results in an M_o-by-N output frame-based signal.

For each output buffer, the block acquires the number of *new* input samples specified by the difference between the **Buffer size** (M_o) and **Buffer overlap** (L) parameters. Each new input sample enters at the bottom of the buffer, and is pushed upwards as later samples enter. The first row in the output therefore corresponds to the earliest input sample. Because the block can buffer a signal to a larger or smaller frame size, the number of samples acquired from the input can be greater or less than the number of samples in an individual input frame.

In general, the output frame period, T_{fo}, is related to the input sample period, T_{si}, by

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

where M_o is the **Output buffer size (per channel)**, and L is the **Buffer overlap**.

As a result, the new output sample period, T_{so}, is

$$T_{so} = \frac{(M_o - L)T_{si}}{M_o}$$

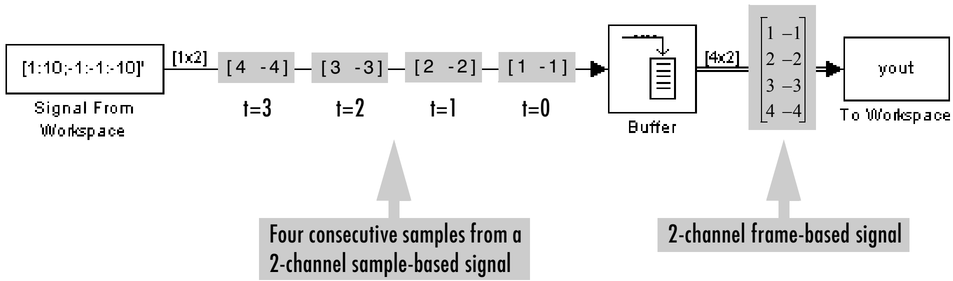
Clearly, this is equal to the input sample period *only* when the **Buffer overlap** is zero. See "Converting Sample Rates and Frame Rates" on page 3-20 for more information about rate conversions.

The following sections provide examples of buffering, and explore related buffering issues:

- "Example: Buffering Sample-Based Signals without Overlap" on page 3-49
- "Overlapping Buffers" on page 3-50
- "Example: Buffering Sample-Based Signals with Overlap" on page 3-50

- "Example: Buffering Frame-Based Signals with Overlap" on page 3-52
- "Buffering Delay and Initial Conditions" on page 3-53

Example: Buffering Sample-Based Signals without Overlap. In the model below, a two-channel sample-based signal is buffered into a two-channel frame-based signal.



To build the model, make the following parameter settings:

- In Signal From Workspace:
  - **Signal** = `[1:10;-1:-1:-10]'`
  - **Sample time** = `1`
  - **Samples per frame** = `1`
- In Buffer
  - **Output buffer size** = `4`
  - **Buffer overlap** = `0`
  - **Initial conditions** = `0`

The Signal From Workspace block generates one two-channel sample at each sample time due to the **Samples per frame** parameter setting of 1. The **Buffer size** setting of `4` in the Buffer block results in a frame-based output with frame size 4.

A much better way to create the frame-based signal shown above is to set the **Samples per frame** parameter of the Signal From Workspace block to `4`. The Signal From Workspace block then performs the buffering internally, and directly generates the two-channel frame-based signal; the separate Buffer block is not needed. See the following sections for more information:

- "Creating Signals Using the Signal From Workspace Block" on page 3-38

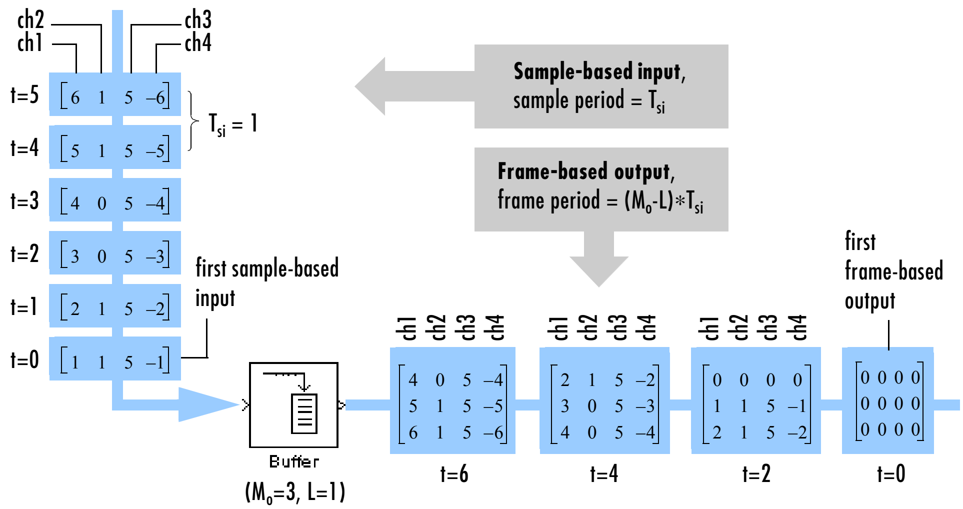
- “Importing a Multichannel Frame-Based Signal” on page 3-68

Overlapping Buffers. In some cases it is useful to work with data that represents overlapping sections of an original sample-based or frame-based signal. In estimating the power spectrum of a signal, for example, it is often desirable to compute the FFT of overlapping sections of data. Overlapping buffers are also needed in computing statistics on a sliding window, or for adaptive filtering. The **Buffer overlap** parameter of the Buffer block specifies the number of overlap points, L.

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

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

Example: Buffering Sample-Based Signals with Overlap. In the following model, 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 overlap, the input sample period is not conserved, and the output sample period is 2/3.



To build the model, define the following variable in the MATLAB workspace.

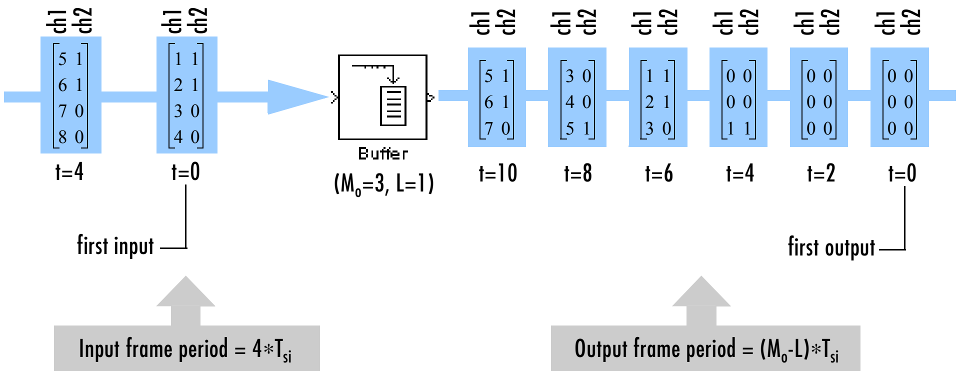
```
A = [1 1 5 -1;2 1 5 -2;3 0 5 -3;4 0 5 -4;5 1 5 -5;6 1 5 -6];
```

Connect the Buffer block to a Signal From Workspace source and a To Workspace sink with the following parameter settings:

- In the Signal From Workspace block, set:
  - **Signal** = A
  - **Sample time** = 1
  - **Samples per frame** = 1
- In the Buffer block, set:
  - **Output buffer size (per channel)** = 3
  - **Buffer overlap** = 1
  - **Initial conditions** = 0

Note that the inputs do not begin appearing at the output until the second row of the second matrix. This is due to the block's latency. See "Delay and Latency" on page 3-85 for general information about algorithmic delay, and see "Buffering Delay and Initial Conditions" on page 3-53 for instructions on how to calculate buffering delay.

Example: Buffering Frame-Based Signals with Overlap. In the model below, a two-channel frame-based signal with frame period 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.



To build the model, define the following variable in the MATLAB workspace.

```
A = [1 1;2 1;3 0;4 0;5 1;6 1;7 0;8 0];
```

Connect the Buffer block to a Signal From Workspace source and a To Workspace sink with the following parameter settings:

- In the Signal From Workspace block, set
  - **Signal** = A
  - **Sample time** = 1
  - **Samples per frame** = 4
- In the Buffer block, set
  - **Output buffer size (per channel)** = 3
  - **Buffer overlap** = 1
  - **Initial conditions** = 0

Note that the inputs do not begin appearing at the output until the last row of the third matrix. This is due to the block's latency. See "Delay and Latency" on page 3-85 for general information about algorithmic delay, and see "Buffering Delay and Initial Conditions" on page 3-53 for instructions on how to calculate buffering delay.

Buffering Delay and Initial Conditions. In both of the previous buffering examples the input signal is delayed by a certain number of samples. In “Example: Buffering Sample-Based Signals with Overlap” the delay is four samples. In “Example: Buffering Frame-Based Signals with Overlap” the delay is eight samples. The initial output samples adopt the value specified for the **Initial condition** parameter, which is zero in both examples above.

In fact, under most conditions the Buffer and Unbuffer blocks have some amount of *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 in multitasking operations, where `f` is the input frame size, `n` is the **Buffer size** parameter setting, and `m` is the **Buffer overlap** parameter setting.

For example, if you had run the frame-based example model in multitasking mode, you could compute the latency by entering the following command at the MATLAB command line:

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

This agrees with the block’s output in that example. See “Delay and Latency” on page 3-85 and the “Latency” section on each block reference page for more information.

Deconstructing Signals

Multichannel signals, represented by matrices in Simulink, are frequently used in DSP models for efficiency and compactness. An M-by-N sample-based multichannel signal represents M*N independent signals (one sample from each), whereas an M-by-N frame-based multichannel signal represents N independent channels (M consecutive samples from each). See “Multichannel Signals” on page 3-11 for more information about the matrix format.

Even though most of the DSP blocks can process multichannel signals, you may sometimes need to access just one channel or a particular range of samples in a multichannel signal. There are a variety of ways to *deconstruct* multichannel signals, the most common of which are explained in the following sections:

- “Deconstructing Multichannel Sample-Based Signals” on page 3-54
- “Deconstructing Multichannel Frame-Based Signals” on page 3-57

For information about constructing multichannel signals from individual sample-based or frame-based signals, see the following sections:

- “Constructing Multichannel Sample-Based Signals” on page 3-42
- “Constructing Multichannel Frame-Based Signals” on page 3-45

Deconstructing Multichannel Sample-Based Signals

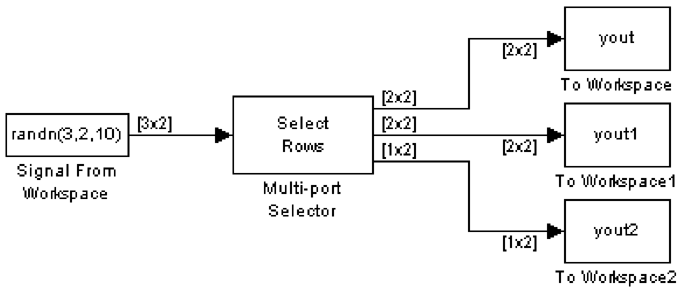
A sample-based signal with M∗N channels is represented by a sequence of M-by-N matrices. (The special case of M = N = 1 represents a single-channel signal.) You can access individual channels of the multichannel signal by using the blocks in the Indexing library (in Signal Management). The following sections explain how to do this:

- “Deconstructing a Sample-Based Multichannel Signal into Multiple Independent Signals” on page 3-54
- “Deconstructing a Sample-Based Multichannel Signal into a Related Multichannel Signal” on page 3-55

Deconstructing a Sample-Based Multichannel Signal into Multiple Independent Signals

You can split a multichannel sample-based signal into individual sample-based signals (single-channel or multichannel) by using the Multiport Selector block in the Indexing library (in Signal Management). Any subset of rows or columns can be selected for propagation to a given output port.

Example: Deconstructing to Independent Signals. In the model below, a six-channel sample-based signal (3-by-2 matrix) is deconstructed to yield three independent sample-based signals. Two of the output signals have four channels, and the third signal has two channels.



To build the model, make the following parameter settings:

- In Signal From Workspace, set **Signal** = `randn(3,2,10)`
- In Multiport Selector, set:
  - **Select** = **Rows**
  - **Indices to output** = `{[1 2],[1 3],3}`

The **Indices to output** setting specifies that rows 1 and 2 of the input should be reproduced at output 1, that rows 1 and 3 of the input should be reproduced at output 2, and that row 3 of the input should be reproduced alone at output 3.

See "Sample-Based Multichannel Signals" on page 3-11 for more about sample-based signals.

Deconstructing a Sample-Based Multichannel Signal into a Related Multichannel Signal

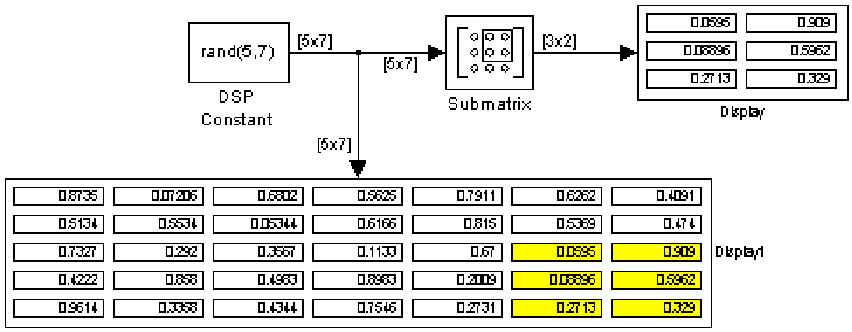
You can select a subset of channels from a multichannel sample-based signal by using one of the following blocks in the Indexing library (in Signal Management):

- Selector (Simulink)
- Submatrix

- Variable Selector

The next section provides an example of using the Submatrix block to extract a portion of a multichannel sample-based signal. The Submatrix block is the most versatile of the above blocks in that it allows you to make completely arbitrary channel selections.

Example: Deconstructing to a Multichannel Signal. In the model below, a 35-channel sample-based signal (5-by-7 matrix) is deconstructed to yield a sample-based signal containing only six of the original channels.



To build the model, make the following parameter settings:

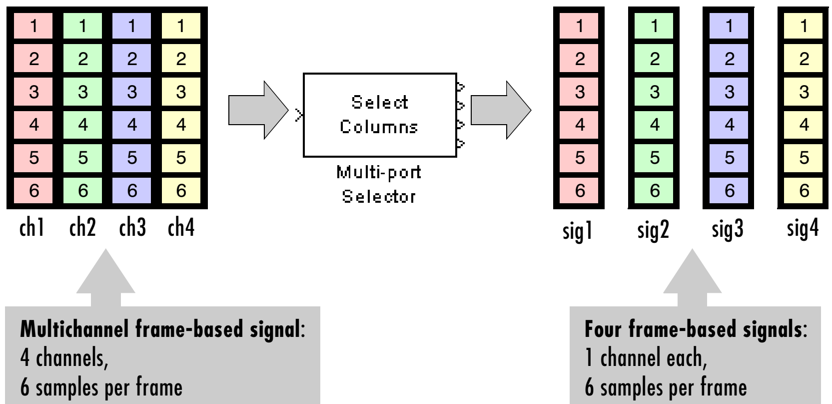
- In DSP Constant, set **Constant value** = `rand(5,7)`
- In Submatrix, set:
  - **Row span** = **Range of rows**
  - **Starting row** = **Index**
  - **Starting row index** = 3
  - **Ending row** = **Last**
  - **Column span** = **Range of columns**
  - **Starting column** = **Offset from last**
  - **Starting column index** = 1
  - **Ending column** = **Last**

See “Sample-Based Multichannel Signals” on page 3-11 for more about sample-based signals.

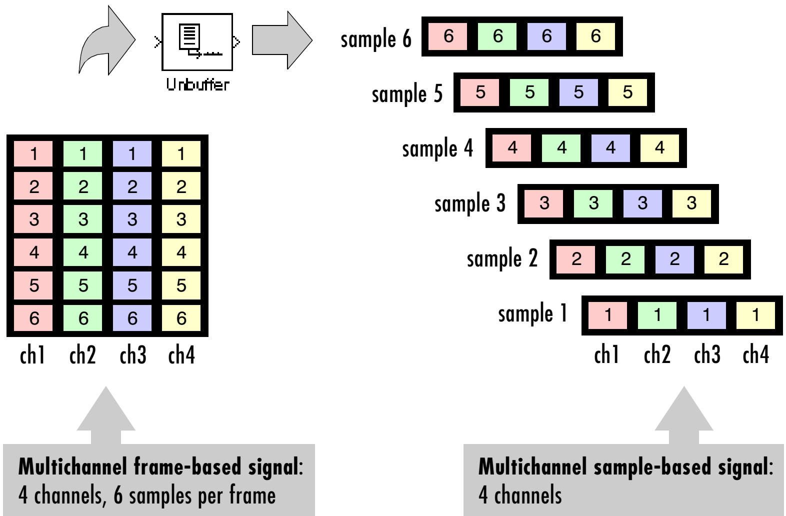
Deconstructing Multichannel Frame-Based Signals

A frame-based signal with N channels and frame size M is represented by a sequence of M-by-N matrices. (The special case of N = 1 represents a single-channel signal.) There are two basic ways to deconstruct a multichannel frame-based signal:

- *Split the channels into independent signals* — The constituent channels of a multichannel frame-based signal can be extracted to form individual frame based signals (with the same frame rate and size) by using the Multiport Selector block in the Indexing library (in Signal Management).



- *Unbuffer the samples* — Multichannel frame-based signals can be unbuffered into multichannel sample-based signals using the Unbuffer block in the Buffers library (in Signal Management).



The following sections explain the two methods of deconstructing multichannel frame-based signals:

- "Splitting a Multichannel Signal into Individual Signals" on page 3-58
- "Unbuffering a Frame-Based Signal into a Sample-Based Signal" on page 3-60

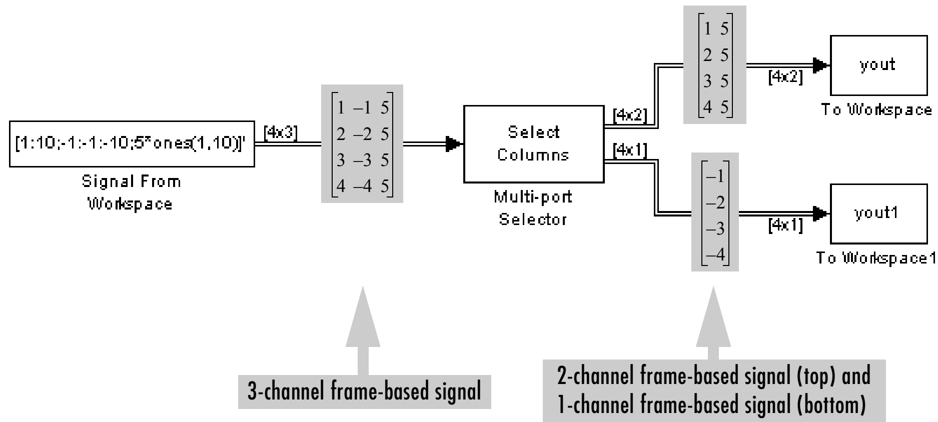
The final section explains how to reorder the channels in a frame-based signal *without* splitting the channels apart:

- "Reordering Channels in a Frame-Based Multichannel Signal" on page 3-61

Splitting a Multichannel Signal into Individual Signals

You can split a frame-based multichannel signal into its constituent frame-based signals by using the Multiport Selector block in the Indexing library (in Signal Management).

Example: Splitting a Multichannel Frame-Based Signal. In the model below, a three-channel frame-based signal is split into a single-channel frame-based signal and a two-channel frame-based signal.



To build the model, make the following parameter settings:

- In Signal From Workspace, set:
  - **Signal** = `[1:10;-1:-1:-10;5*ones(1,10)]'`
  - **Samples per frame** = `4`
- In Multiport Selector, set:
  - **Select** = **Columns**
  - **Indices to output** = `{[1 3],2}`

The top (4-by-2) output from the Multiport Selector block contains the first and third input channels, and the bottom output contains the second input channel. The Multiport Selector block preserves the frame rate and frame size of the input as long as **Select** is set to **Columns**. See "Frame-Based Multichannel Signals" on page 3-12 for more about frame-based signals.

Note that you could also create or import the two signals by using two distinct Signal From Workspace blocks. See the following sections for more information:

- "Creating Signals Using the Signal From Workspace Block" on page 3-38
- "Importing a Multichannel Frame-Based Signal" on page 3-68

Unbuffering a Frame-Based Signal into a Sample-Based Signal

You can unbuffer a multichannel frame-based signal into a multichannel sample-based signal by using the Unbuffer block in the Buffers library (in Signal Management).

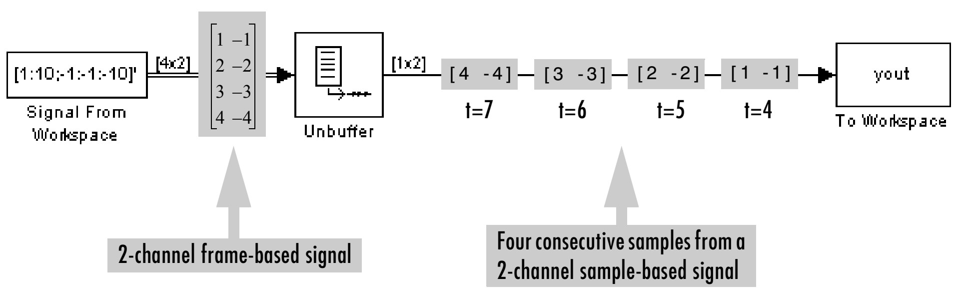
The Unbuffer block performs the inverse operation of the Buffer block's "sample-based to frame-based" buffering process, and generates an N-channel sample-based output from an N-channel frame-based input. The first row in each input matrix is always the first sample-based output. In other words, the Unbuffer block unbuffers each input frame from the top down.

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 Rates and Frame Rates" on page 3-20 for more information about rate conversions.

Example: Unbuffering a Frame-Based Signal. In the model below, a two-channel frame-based signal is unbuffered into a two-channel sample-based signal.



To build the model, make the following parameter settings:

- In Signal From Workspace:
  - **Signal** = `[1:10;-1:-1:-10]'`
  - **Sample time** = `1`
  - **Samples per frame** = `4`

The Signal From Workspace block generates a two-channel frame based-signal with frame size 4 (because the **Samples per frame** parameter is set to 4). The Unbuffer block unbuffers this signal to a two-channel sample-based signal.

Note The Unbuffer block generates initial conditions (not shown in the figure above) 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.

Reordering Channels in a Frame-Based Multichannel Signal

Use the Permute Matrix block to swap channels in a frame-based signal.

Importing Signals

Although a number of signal generation blocks are available in Simulink and the DSP Blockset, it is very common to import custom signals from the MATLAB workspace as well. The following sections explain how to do this:

- "Importing a Multichannel Sample-Based Signal" on page 3-62
- "Importing a Multichannel Frame-Based Signal" on page 3-68
- "Importing WAV Files" on page 3-71

For information about creating signals, see the following sections:

- "Creating Signals Using Constant Blocks" on page 3-33
- "Creating Signals Using Signal Generator Blocks" on page 3-36
- "Creating Signals Using the Signal From Workspace Block" on page 3-38

Importing a Multichannel Sample-Based Signal

The Signal From Workspace block in the DSP Sources library is the key block for importing sample-based signals of all dimensions from the MATLAB workspace.

Signal From
Workspace

The dialog box has the following parameters:

- **Signal**
- **Sample time**
- **Samples per frame**
- **Form output after final data value by**

Use the **Signal** parameter to specify the name of a variable (vector, matrix, or 3-D array) in the MATLAB workspace. You can also enter any valid MATLAB expressions involving workspace variables, as long as the expressions evaluate to a vector, matrix, or 3-D array.

The **Samples per frame** parameter must be set to 1 for sample-based output; any value larger that 1 produces a frame-based output. See "Importing a Multichannel Frame-Based Signal" on page 3-68 for more information. The

Sample-time parameter specifies the sample period of the sample-based output. See "Sample-Based Multichannel Signals" on page 3-11 for general information about sample-based signals.

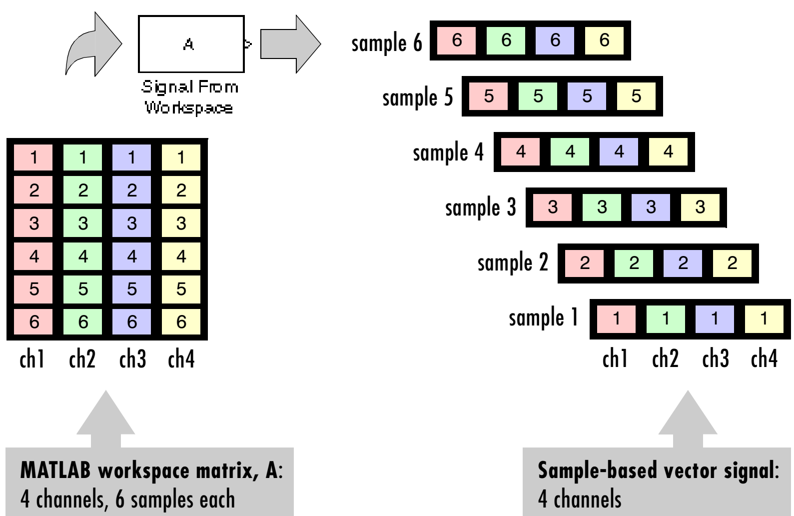
The following sections explain how the Signal From Workspace generates its output:

- "Importing a Sample-Based Vector Signal" on page 3-63
- "Importing a Sample-Based Matrix Signal" on page 3-65

Importing a Sample-Based Vector Signal

The Signal From Workspace block generates a sample-based vector signal when the variable (or expression) in the **Signal** parameter is a matrix and **Samples per frame** = 1. Beginning with the first row of the matrix, the block releases a single row of the matrix to the output at each sample time. Therefore, if the **Signal** parameter specifies an M-by-N matrix, the output of the Signal From Workspace block is a 1-by-N matrix (row vector), representing N channels.

The figure below illustrates this for a 6-by-4 workspace matrix, A.



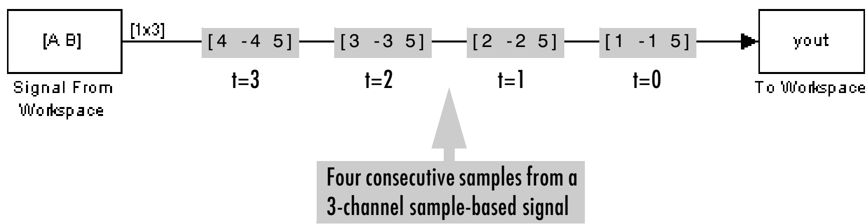
As the figure above suggests, the output of the Signal From Workspace block can only be a valid sample-based signal (having N independent channels) if the M-by-N workspace matrix A in fact represents N independent channels, each containing M consecutive samples. In other words, the workspace matrix must be oriented so as to have the independent channels as its columns.

When the block has output all of the rows available in the specified variable, 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. This behavior is controlled by the **Form output after final data value by** parameter. See the Signal From Workspace reference page for more information.

The following example illustrates how the Signal From Workspace block can be used to import a sample-based vector signal into a model.

Example: Importing a Sample-Based Vector Signal. In the model below, the Signal From Workspace creates a three-channel sample-based signal with the following data:

- **Channel 1**: 1, 2, 3, 4, 5,..., 100, 0, 0, 0,...
- **Channel 2**: -1, -2, -3, -4, -5,..., -100, 0, 0, 0,...
- **Channel 3**: 5, 5, 5, 5, 5,..., 0, 0, 0,...



To create the model, define the following variables at the MATLAB command line

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

Matrix `A` represents a two-channel signal with 100 samples, and matrix `B` represents a one-channel signal with 100 samples.

Specify the following parameter values in the Signal From Workspace block:

- **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. Equivalently, you could set **Signal** = C, and define C at the command line by

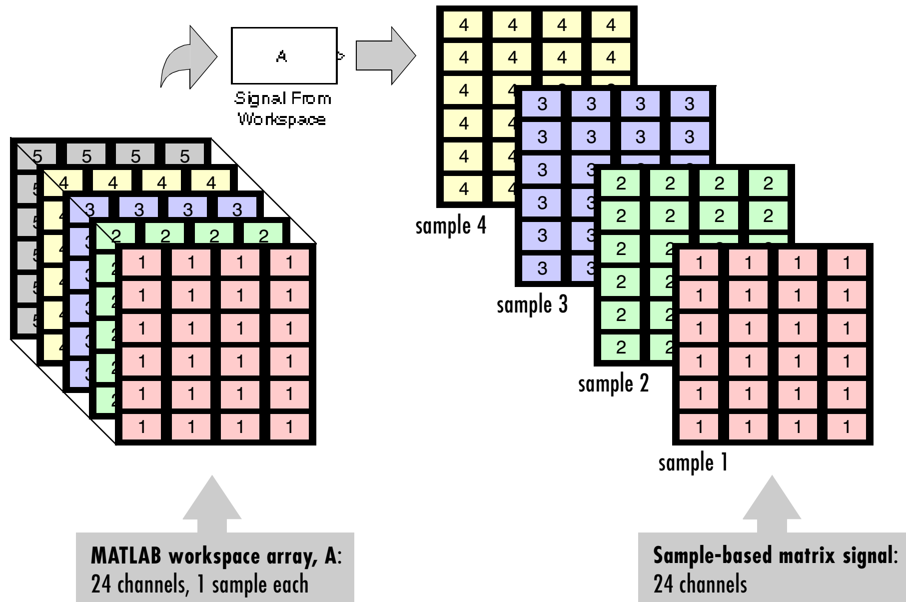
```
C = [A B]
```

The **Sample time** setting of 1 yields a sample-based output with sample period of 1 second. The **Form output after final data value** parameter setting specifies that all outputs after the third are zero.

Importing a Sample-Based Matrix Signal

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 **Samples per frame** = 1. Beginning with the first page of the array, the block releases a single page (i.e., matrix) 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 an M-by-N matrix, representing M∗N channels.

The figure below illustrates this for a 6-by-4-by-5 workspace array A.



As the figure above suggests, the output of the Signal From Workspace block can only be a valid sample-based signal (having M∗N independent channels) if the M-by-N-by-P workspace array A in fact represents M∗N independent channels, each having P samples. In other words, the workspace array must be oriented to have time running along its third (P) dimension.

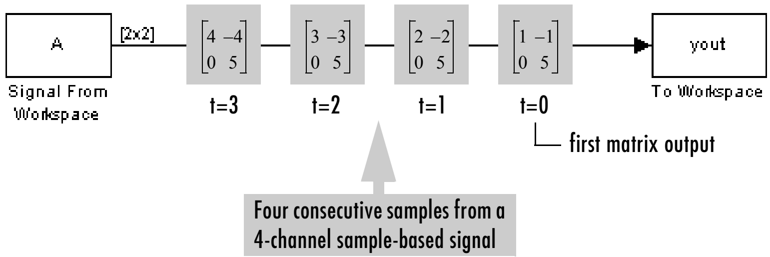
When the block has output all of the pages available in the specified array, it can start again at the beginning of the array, or simply repeat the final page (or generate zero-matrices) until the end of the simulation. This behavior is controlled by the **Form output after final data value by** parameter. See the Signal From Workspace reference page for more information.

The following example illustrates how the Signal From Workspace block can be used to import a sample-based matrix signal into a model.

Example: Importing a Sample-Based Matrix Signal. In the model below, the Signal From Workspace imports a four-channel sample-based signal with the following data:

- **Channel 1**: 1, 2, 3, 4, 5,..., 100, 0, 0, 0,...
- **Channel 2**: -1, -2, -3, -4, -5,..., -100, 0, 0, 0,...

- **Channel 3**: 0, 0, 0, 0, 0,...
- **Channel 4**: 5, 5, 5,..., 0, 0, 0,...



To create the model, define the following variables at the MATLAB command line.

```
sig1 = reshape(1:100,[1 1 100])         % 1-by-1-by-100 array
sig2 = reshape(-1:-1:-100,[1 1 100])    % 1-by-1-by-100 array
sig3 = zeros(1,1,100)                   % 1-by-1-by-100 array
sig4 = 5*ones(1,1,100)                  % 1-by-1-by-100 array
sig12 = cat(2,sig1,sig2)                % 1-by-2-by-100 array
sig34 = cat(2,sig3,sig4)                % 1-by-2-by-100 array

A = cat(1,sig12,sig34)                  % 2-by-2-by-100 array
```

Array A represents a 4-channel signal with 100 samples.

Specify the following parameter values in the Signal From Workspace block:

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

The **Sample time** and **Samples per frame** settings of 1 yield a sample-based output with sample period of 1 second. Each of the four elements in the matrix represents an independent channel. The **Form output after final data value** parameter setting specifies that all outputs after the one-hundredth are zero.

These two sections may also be of interest to you:

- "Creating Signals Using the Signal From Workspace Block" on page 3-38

• "Constructing Multichannel Sample-Based Signals" on page 3-42

Importing a Multichannel Frame-Based Signal

The Signal From Workspace block in the DSP Sources library is the key block for importing frame-based signals from the MATLAB workspace.

The dialog box has the following parameters:

- **Signal**
- **Sample time**
- **Samples per frame**
- **Form output after final data value by**

Use the **Signal** parameter to specify the name of a variable (vector or matrix) in the MATLAB workspace. You can also enter any valid MATLAB expressions involving workspace variables, as long as the expressions evaluate to a vector or matrix.

The **Samples per frame** parameter must be set to a value greater than 1 for frame-based output; a value of 1 produces sample-based output. See "Importing a Multichannel Sample-Based Signal" on page 3-62 for more information.

The **Sample-time** parameter specifies the sample period, T_s, of the frame-based output. The frame period of the signal is $M*T_s$, where M is the value of the **Samples per frame** parameter. See "Frame-Based Multichannel Signals" on page 3-12 for general information about frame-based signals.

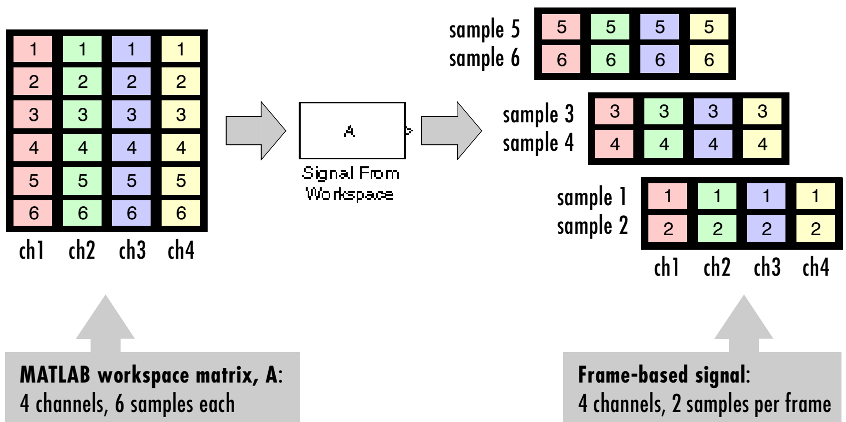
The following section explains how the Signal From Workspace block generates its frame-based output.

Importing a Frame-Based Signal with the Signal From Workspace Block

The Signal From Workspace block generates a frame-based multichannel signal when the variable (or expression) in the **Signal** parameter is a matrix, and the **Samples per frame** parameter specifies a value M greater than 1.

Beginning with the first M rows of the matrix, the block releases M rows of the matrix (i.e., one frame from each channel) to the output every $M*T_s$ seconds. Therefore, if the **Signal** parameter specifies a W-by-N workspace matrix, the output of the Signal From Workspace block is an M-by-N matrix representing N channels.

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



As the figure above suggests, the output of the Signal From Workspace block can only be a valid frame-based signal (having N independent channels) if the W-by-N workspace matrix A in fact represents N independent channels. In other words, the workspace matrix must be oriented so as to have the independent channels as its columns.

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.

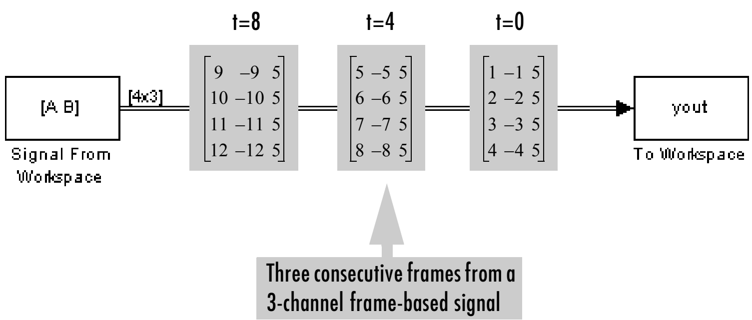
When the block has output all of the rows available in the specified variable, it can start again at the beginning of the signal, or simply repeat the final value (or generate zeros) until the end of the simulation. This behavior is controlled by the **Form output after final data value by** parameter. See the Signal From Workspace block reference page for more information.

The following example illustrates how the Signal From Workspace block is used to import a frame-based multichannel signal into a model.

Example: Importing a Frame-Based Signal. In the model below, the Signal From Workspace block creates a three-channel frame-based signal with the following data:

- **Channel 1**: 1, 2, 3, 4, 5,..., 100, 0, 0, 0,...
- **Channel 2**: -1, -2, -3, -4, -5,..., -100, 0, 0, 0,...
- **Channel 3**: 5, 5, 5, 5, 5,..., 0, 0, 0,...

The frame size is four samples.



To create the model, define the following variables at the MATLAB command line.

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

Matrix `A` represents a two-channel signal with 100 samples, and matrix `B` represents a one-channel signal with 100 samples.

Specify the following parameter values in the Signal From Workspace block:

- **Signal** = `[A B]`
- **Sample time** = `1`
- **Samples per frame** = `4`
- **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`. Equivalently, you could set **Signal** = `C`, and define `C` at the command line by

```
C = [A B]
```

The **Sample time** setting of `1` and **Samples per frame** setting of `4` yield a frame-based output with sample period of 1 second and frame period of 4 seconds. The **Form output after final data value** parameter setting specifies that all samples after the hundredth are zero.

Importing WAV Files

The key blocks for importing WAV audio files are

- From Wave Device
- From Wave File

See the online reference pages for complete information.

Exporting Signals

The To Workspace and Triggered To Workspace blocks are the primary conduits for exporting signals from a Simulink model to the MATLAB workspace. The following sections explain how to use these important blocks:

- "Exporting Multichannel Signals" on page 3-72
- "Exporting and Playing WAV Files" on page 3-79

Exporting Multichannel Signals

The To Workspace block in the Simulink Sources library is the key block for exporting signals of all dimensions to the MATLAB workspace.

The dialog box has the following parameters:

- **Variable name**
- **Limit data points to last**
- **Decimation**
- **Sample time**
- **Save format**

Use the **Variable name** parameter to specify the workspace variable in which the output should be saved. (An existing output with the same name is overwritten.)

The **Limit data points to last** parameter specifies how many of the most recent output samples should be retained in the specified workspace variable. For example, if you specify **Limit data points to last** = `100`, then even if the simulation propagates thousands of samples to the To Workspace block, only the most recent 100 samples will actually be saved in the workspace. By setting a limit on the number of saved samples, you can prevent out-of-memory errors for long-running simulations. Note, however, that the default setting for **Limit data points to last** is `inf`, which allows the workspace variable to grow indefinitely large.

The default values of `1` and `-1` for the **Decimation** and **Sample time** parameters (respectively) are generally adequate for DSP models. If you want to downsample a signal before exporting to the workspace, consider using the Downsample or FIR Decimation blocks. See “Converting Sample Rates and Frame Rates” on page 3-20 for more information about rate conversion.

The **Save format** parameter allows you to save the output in a variety of formats. The default is **Array**, which is also generally the most accessible output format. Although this format does not save a record of the sample times corresponding to the output samples, you can create such a record for a given model by selecting the **Time** option in the **Workspace I/O** panel of the **Simulation Parameters** dialog box. See “Performance-Related Settings in dspstartup.m” on page C-4 for more information.

The following sections explain how the To Workspace block generates its output:

- “Exporting a Sample-Based Signal Using the To Workspace Block” on page 3-73
- “Exporting a Frame-Based Signal Using the To Workspace Block” on page 3-76

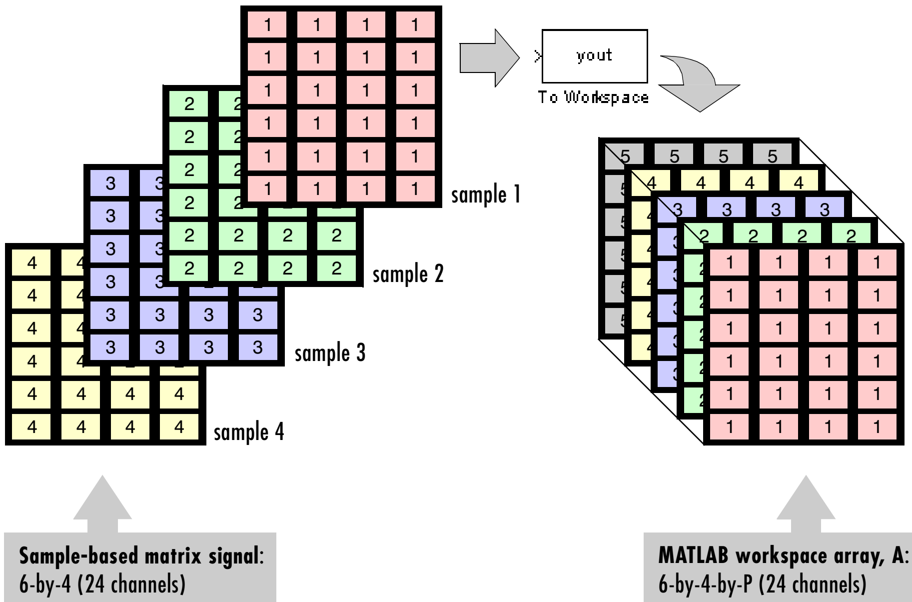
The following two sections may also be of interest:

- “Creating Signals Using the Signal From Workspace Block” on page 3-38
- “Constructing Multichannel Sample-Based Signals” on page 3-42

Exporting a Sample-Based Signal Using the To Workspace Block

Recall that a sample-based signal with M*N channels is represented by a sequence of M-by-N matrices. (The special case of M = N = 1 represents a single-channel signal.) When the input to the To Workspace block is a sample-based signal (and the **Save format** parameter is set to **Array**), 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 illustrates this for a 6-by-4 sample-based signal exported to workspace array `A`.



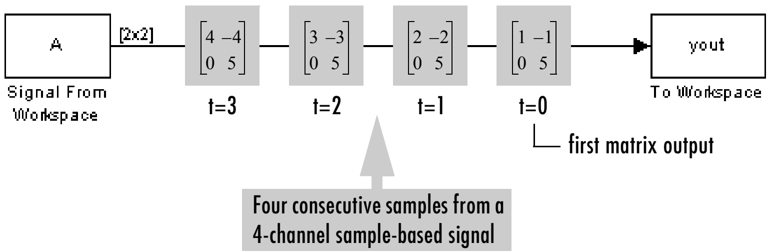
The workspace array always has time running along its third (P) dimension. Samples are saved along the P dimension whether the input is a matrix, vector, or scalar (single channel).

The following example illustrates how the To Workspace block can be used to export a sample-based matrix signal to the MATLAB workspace.

Example: Exporting a Sample-Based Matrix Signal. In the model below, the To Workspace block exports a four-channel sample-based signal with the following data:

- **Channel 1**: 1, 2, 3, 4, 5,..., 100, 0, 0, 0,...
- **Channel 2**: -1, -2, -3, -4, -5,..., -100, 0, 0, 0,...
- **Channel 3**: 0, 0, 0, 0, 0,...
- **Channel 4**: 5, 5, 5,..., 0, 0, 0,...

The first four consecutive samples are shown in the figure.



To create the model, define the following variables at the MATLAB command line.

```
sig1 = reshape(1:100,[1 1 100])        % 1-by-1-by-100 array
sig2 = reshape(-1:-1:-100,[1 1 100])   % 1-by-1-by-100 array
sig3 = zeros(1,1,100)                  % 1-by-1-by-100 array
sig4 = 5*ones(1,1,100)                 % 1-by-1-by-100 array
sig12 = cat(2,sig1,sig2)               % 1-by-2-by-100 array
sig34 = cat(2,sig3,sig4)               % 1-by-2-by-100 array

A = cat(1,sig12,sig34)                 % 2-by-2-by-100 array
```

Array `A` represents a four-channel signal with 100 samples.

Specify the following parameter values in the Signal From Workspace block:

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

Specify the following parameter values in the To Workspace block:

- **Variable name** = `yout`
- **Limit data points to last** = `inf`
- **Decimation** = `1`
- **Sample time** = `-1`
- **Save format** = **Array**

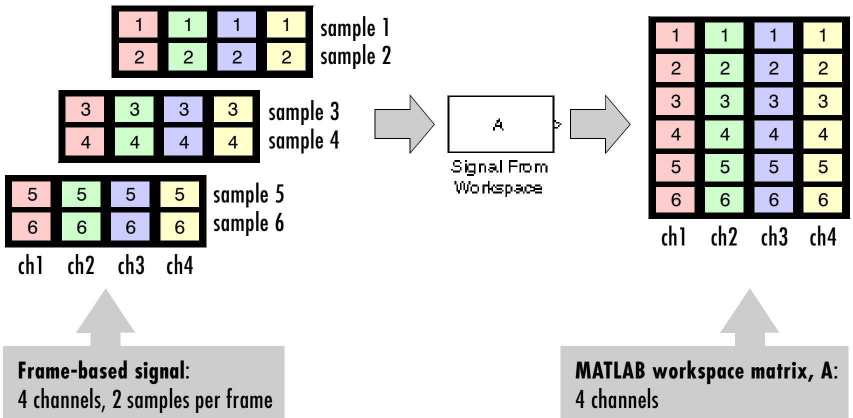
Run the model, and look at output `yout`. The first four samples (pages) are shown below:

```
yout(:,:,1:4)

ans(:,:,1) =

     1    -1
     0     5

ans(:,:,2) =

     2    -2
     0     5

ans(:,:,3) =

     3    -3
     0     5

ans(:,:,4) =

     4    -4
     0     5
```

Exporting a Frame-Based Signal Using the To Workspace Block

Recall that a frame-based signal with N channels and frame size M is represented by a sequence of M-by-N matrices. (The special case of N = 1 represents a single-channel signal.) When the input to the To Workspace block is a frame-based signal (and the **Save format** parameter is set to **Array**), 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 figure below illustrates this for three consecutive frames of a frame-based signal (two samples per frame) exported to matrix A.

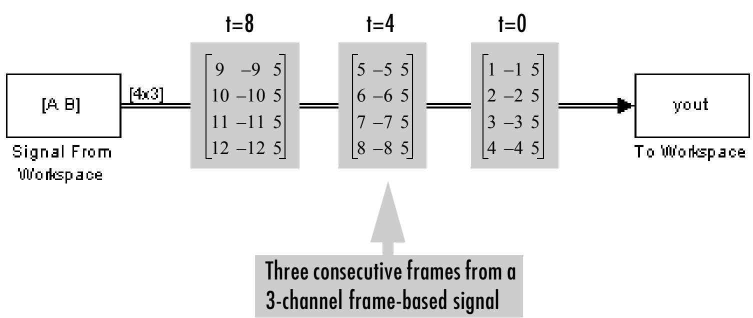


The workspace matrix always has time running along its first (P) dimension. Samples are saved along the P dimension whether the input is a matrix, vector, or scalar (single channel).

The following example illustrates how the To Workspace block can be used to export a frame-based multichannel signal to the MATLAB workspace.

Example: Exporting a Frame-Based Signal. In the model below, the To Workspace block exports a three-channel frame-based signal with the following data:

- **Channel 1**: 1, 2, 3, 4, 5,..., 100, 0, 0, 0,...
- **Channel 2**: -1, -2, -3, -4, -5,..., -100, 0, 0, 0,...
- **Channel 3**: 5, 5, 5, 5, 5,..., 0, 0, 0,...



To create the model, define the following variables at the MATLAB command line:

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

Matrix `A` represents a two-channel signal with 100 samples, and matrix `B` represents a one-channel signal with 100 samples.

Specify the following parameter values in the Signal From Workspace block:

- **Signal** = `[A B]`
- **Sample time** = `1`
- **Samples per frame** = `4`
- **Form output after final data value** = **Setting to zero**

The **Sample time** setting of `1` and **Samples per frame** setting of `4` yield a frame-based output with sample period of 1 second and frame period of 4 seconds.

Specify the following parameter values in the To Workspace block:

- **Variable name** = `yout`
- **Limit data points to last** = `inf`
- **Decimation** = `1`
- **Sample time** = `-1`
- **Save format** = **Array**

Run the model, and look at output `yout`. The first 10 samples (rows) are shown:

```
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
```

These two sections may also be of interest to you:

- “Creating Signals Using the Signal From Workspace Block” on page 3-38
- “Constructing Multichannel Sample-Based Signals” on page 3-42

Exporting and Playing WAV Files

The key blocks for exporting and playing WAV audio files are

- To Wave Device
- To Wave File

The To Wave Device and To Wave File blocks are limited to one-channel (mono) or two-channel (stereo) inputs, selectable in the **Stereo** check box. See the reference pages for complete information.

The following demos may also be of interest:

- Audio Flanger — PC/Windows
- Demonstration of Audio Reverberation
- Basic LPC Speech Coding - PC/Windows

Viewing Signals

The following blocks in the DSP Sinks library are the key blocks for displaying signals:

- Matrix Viewer
- Spectrum Scope
- Time Scope (Simulink Scope)
- Vector Scope

The following sections provide an introduction to how these blocks are commonly used:

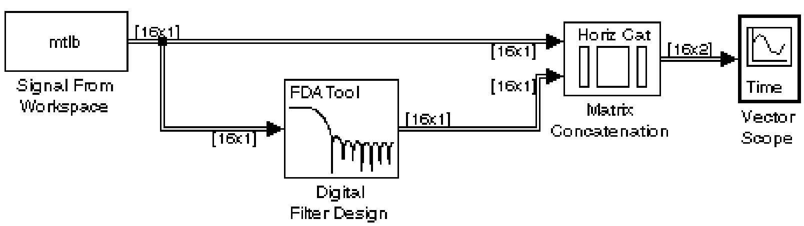
- "Displaying Signals in the Time-Domain" on page 3-80
- "Displaying Signals in the Frequency-Domain" on page 3-82
- "Displaying Matrices" on page 3-83

Displaying Signals in the Time-Domain

The Vector Scope block can display both time-domain and frequency-domain data. It differs from the Spectrum Scope in that it does not compute the FFT of inputs.

Example: Displaying Time-Domain Data

In the model below, two frame-based signals are simultaneously displayed on the scope.



To create the model, first load the `mtlb` signal:

```
load mtlb          % Contains variables 'mtlb' and 'Fs'
```

Specify the following parameter values in the Signal From Workspace block:

- **Signal** = `mtlb`
- **Sample time** = `1`
- **Samples per frame** = `16`
- **Form output after final data value** = **Cyclic Repetition**

Specify the following parameter values in the Digital Filter Design block:

- **Filter Type** = **Lowpass**
- **Design Method** = **FIR** (**Window**)
- **Filter Order** (**Specify order**) = `22`
- **Window Specifications** (**Window**) = **Hamming**
- **Frequency Specifications** (**wc**) = `0.25`
- **Frequency Specifications** (**Units**) = **Normalized (0 to 1)**
- **Magnitude Specifications** (**Units**) = **dB**

Specify the following parameter values in the **Scope properties** pane of the Vector Scope block:

- **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. You may want to set the **Stop time** in the **Simulation Parameters** dialog box to `inf` to allow the simulation to run longer. The following section provides a few tips for improving the display.

Improving the Appearance of the Display. You may want to alter the appearance of the scope display by making some of the following adjustments from the right-click popup menu. To access the right-click menu, click with the right mouse button anywhere in the plot region. These options are also available from the **Axes** and **Channels** menus that are visible at the top of the window when **Compact display** is not selected. You can make all of these changes while the simulation is running:

- Select **Autoscale** at any time from the right-click menu to rescale the vertical axis to best fit the most recently displayed data.

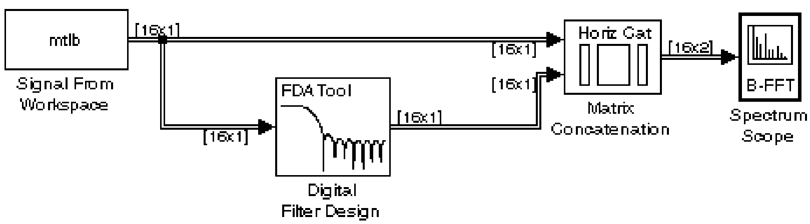
- Select **Compact display** from the right-click menu to allow the scope to use all the available space in the window.
- Select **CH 1** from the right-click menu, and then select **Marker** and "**o**" from the submenus, to mark the data points on the channel 1 signal with circles.
- Select **CH 1** from the right-click menu, and then select **Color** and **Blue** from the submenus, to code the channel 1 signal with the color blue.
- Select **CH 2** from the right-click menu, and then select **Marker** and **Diamond** from the submenus, to mark the data points on the channel 2 signal with diamonds.

Displaying Signals in the Frequency-Domain

The Spectrum Scope block can display the frequency spectra of time-domain input data. It differs from the Vector Scope by computing the FFT of inputs to transform them to the frequency domain.

Example: Displaying Frequency-Domain Data

In the model below, the frequency content of two frame-based signals is simultaneously displayed on the scope.



To create the model, first load the `mtlb` signal:

```
load mtlb          % Contains variables 'mtlb' and 'Fs'
```

Specify the following parameter values in the Signal From Workspace block:

- **Signal** = `mtlb`
- **Sample time** = 1
- **Samples per frame** = 16
- **Form output after final data value** = **Cyclic Repetition**

Specify the following parameter values in the Digital Filter Design block:

- **Filter Type** = **Lowpass**
- **Design Method** = **FIR** (**Window**)
- **Filter Order** (**Specify order**) = 22
- **Window Specifications** (**Window**) = **Hamming**
- **Frequency Specifications** (**wc**) = `0.25`
- **Frequency Specifications** (**Units**) = **Normalized (0 to 1)**
- **Magnitude Specifications** (**Units**) = **dB**

Specify the following parameter values in the **Scope properties** pane of the Spectrum Scope block:

- **Buffer input** = ☑
- **Buffer size** = 128
- **Buffer overlap** = 64
- **Specify FFT length** = ☐
- **Number of spectral averages** = 2

With these settings, 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.

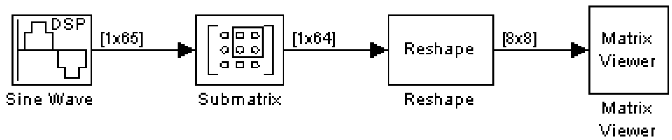
You may want to set the **Stop time** in the **Simulation Parameters** dialog box to `inf` to allow the simulation to run longer. See “Improving the Appearance of the Display” on page 3-81 for some tips on improving the scope display.

Displaying Matrices

The Matrix Viewer block provides general matrix display capabilities that can be used with all matrix signals (frame-based and sample-based).

Example: Displaying Matrices

In the model below, a matrix of shifted sinusoids is displayed with the Matrix Viewer block.



To build the model, specify the following parameter values in the Sine Wave block:

- **Amplitude** = `1`
- **Frequency** = `100`
- **Phase offset** = `0:pi/64:pi`

Specify the following parameter values in the Submatrix block:

- **Row span** = **All rows**
- **Column span** = **Range of columns**
- **Starting column** = **First**
- **Ending column** = **Offset from last**
- **Ending column offset** = `1`

Specify the following parameter values in the Reshape block:

- **Output dimensionality** = **Customize**
- **Output dimensions** = `[8,8]`

Specify **Colormap matrix** = `bone(256)` in the **Image properties** pane of the Matrix Viewer block.

When you run the model, the Matrix Viewer displays each 8-by-8 matrix as it is received. The 256 shades in the specified `bone` colormap are mapped to the range of values specified by the **Minimum input value** and **Maximum input value** parameters; see `colormap` for more information. In this example, these values are `-1.0` and `1.0` respectively, which are appropriate for the sinusoids of amplitude 1 that compose the input signal.

Delay and Latency

There are two distinct types of delay that affect Simulink models:

• "Computational Delay" on page 3-85
• "Algorithmic Delay" on page 3-86

The following sections explain how you can configure Simulink to minimize both varieties of delay and increase simulation performance.

Computational Delay

The *computational delay* of a block or subsystem is related to the number of operations involved in executing that component. 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 (as measured in a benchmark test, for example) depends heavily on the performance of both the computer hardware and underlying software layers, such as MATLAB and the operating system. Computational delay for a particular model therefore typically varies 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 time t=25.0, regardless of the number of operations performed by the FFT algorithm. The Simulink timer reflects only algorithmic delay (described below), not computational delay.

The next section discussed methods of reducing computational delay.

Reducing Computational Delay

There are a number of ways to reduce computational delay without actually running the simulation on faster hardware. To begin with, you should familiarize yourself with "Improving Simulation Performance and Accuracy" in the Simulink documentation, which describes some basic strategies. The section below supplements that information with 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 3-14 for more information.
- Use the `dspstartup` file to tailor Simulink for DSP models, or manually make the adjustments described in "Performance-Related Settings in dspstartup.m" in Appendix C.
- 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 the frame scopes (Vector Scope, etc.).
- Use the Real-Time Workshop to generate generic real-time (GRT) code targeted to your host platform, and simulate 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 Chapter 5, "DSP Block Reference," and elsewhere in this guide, the algorithmic delay of a block is referred to simply as the block's *delay*. It is generally expressed in terms of the number of samples by which a block's output lags behind the corresponding input. This delay is directly related to the time elapsed on the Simulink timer during that block's execution.

The algorithmic delay of a particular block may depend on both the block's parameter settings and the general Simulink settings. To simplify matters, it is helpful to categorize a block's delay using the following levels:

- Zero algorithmic delay
- Basic algorithmic delay
- Excess algorithmic delay (tasking latency)

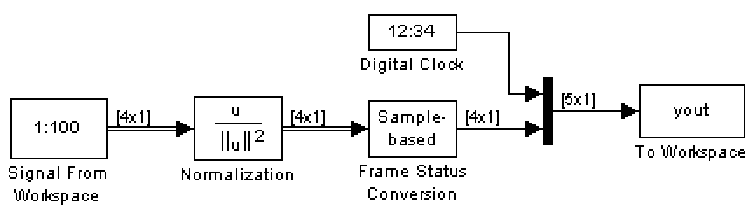
The following sections explain the different levels of delay, and how the simulation and parameter settings can affect the level of delay that a particular block experiences.

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 Chapter 5, "DSP Block Reference," blocks are assumed to have zero delay unless otherwise indicated. In cases where a block has zero delay for one combination of parameter settings but nonzero delay for another, this is noted on the block's reference page.

Example: Zero Algorithmic Delay. Create the model below to observe the operation of the zero-delay Normalization block.



Use the default settings for the Normalization, Digital Clock, Mux, and To Workspace blocks, and adjust the Signal From Workspace block parameters as follows:

- **Signal** = `1:100`
- **Sample time** = `1/4`
- **Samples per frame** = `4`

Select **Sample-based** from the **Output signal** menu in the Frame Status Conversion block.

Note that the current value of the Simulink timer (from the Digital Clock block) is prepended to each output frame. The frame-based signal is converted to a sample-based signal by the Frame Status Conversion so that the output in the command window will be more easily readable.

In the example, the Signal From Workspace block generates a new frame containing four samples once every second (T_{fo} = ¼∗4). The first few output frames are shown below.

(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]'`

When you run the simulation, the normalized output, `yout`, is saved in a workspace array. To convert the array to an easier-to-read matrix format, type

```
squeeze(yout)'
```

The first few samples of the result, `ans`, are shown below:

```
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
```

time

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,

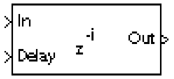
```
[1 2 3 4]' ./ sum([1 2 3 4]'.^2)
```

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 an Integer Delay block with **Delay** > `0`). See the Simulink documentation for more information about algebraic loops.

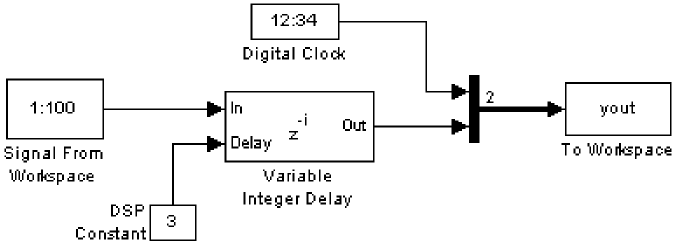
Basic Algorithmic Delay

A typical example of a block that *does* have algorithmic delay is the Variable Integer Delay block.



The input to the `Delay` port of the 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. For example, if the input to the `Delay` port is a constant `3`, and the sample period at both ports is `1`, then a sample that arrives at the block's `In` port at time t=0 is released to the output at time t=3.

Example: Basic Algorithmic Delay. Create the model shown below to observe the operation of a block with basic delay.



Use the default settings for the Digital Clock, Mux, and To Workspace blocks, and adjust the Signal From Workspace block's parameters to the values below:

- **Signal** = `1:100`
- **Sample time** = `1`
- **Samples per frame** = `1`

Set the DSP Constant block's **Constant value** parameter to `3`, and set the Variable Integer Delay block's **Initial conditions** parameter to `-1`.

Now run the simulation and look at the output, `yout`. The first few samples are shown below:

```
yout =

     0    -1
     1    -1
     2    -1
     3     1
     4     2
     5     3
```

time

The first column of `yout` is the Simulink time provided by the Digital Clock block, and 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 DSP 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 fixed at 0.

Consult the online block reference, "Blocks — By Category," for the delay characteristics of particular DSP 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
- Block rate type
- Model rate type
- Block sample mode

Simulink Tasking Mode. Simulink has two tasking modes:

- Single-tasking
- Multitasking

Select a mode by choosing **SingleTasking** or **MultiTasking** from the **Mode** pop-up menu in the **Solver** panel of the **Simulation Parameters** dialog box. The **Mode** pop-up menu is only available when the **Fixed-step** option is selected from the **Type** pop-up menu. (When the **Variable-step** option is selected from the **Type** pop-up menu, Simulink always operates in single-tasking mode.) The **Auto** option in the **Mode** pop-up menu automatically selects single-tasking operation if the model is single-rate (see below), or multitasking operation if the model is multirate.

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 "The Simulation Parameters Dialog Box" in the Simulink documentation for more information about the tasking modes and other simulation options.

Block Rate Type. A block is called *single-rate* when all of its input and output ports operate at the same frame rate (as indicated by identical Probe block measurements or sample time color coding on the input and output lines). 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, which 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. For more information about algebraic loops, see “Algebraic Loops” in the Simulink documentation.

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.

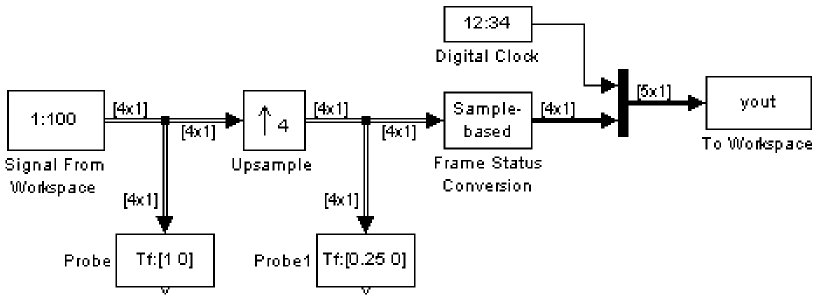
Block Sample Mode. Many blocks can operate in either sample-based or frame-based modes. In source blocks, the mode is usually determined by the **Samples per frame** parameter; a value of 1 for this parameter indicates sample-based mode, while a value greater than 1 indicates frame-based mode. In nonsource blocks, the sample mode is determined by the input signal. See the online block reference for additional information on particular blocks.

Predicting Tasking Latency

The specific amount of tasking latency created by a particular combination of block parameter and simulation settings is described in the “Latency” section of the reference page for the block in question. The following examples show how to use the online block reference to predict tasking latency:

- “Example: Nonzero Tasking Latency” on page 3-93
- “Example: Zero Tasking Latency” on page 3-95

Example: Nonzero Tasking Latency. Most multirate blocks experience tasking latency only in the Simulink multitasking mode. As an example, consider the following model.



To engage the Simulink multitasking mode, adjust the following settings in the **Solver** panel of the **Simulation Parameters** dialog box:

- **Type** = **Fixed-step**
- **Mode** = **MultiTasking**

Use the default settings for the Mux and To Workspace blocks. Adjust the other blocks' parameter settings as follows:

- Set the Signal From Workspace block's parameters to the values below.
  - **Signal** = `1:100`
  - **Sample time** = `1/4`
  - **Samples per frame** = `4`
- Set the Upsample block's parameters to the values below. The **Maintain input frame size** setting of the **Frame-based mode** parameter makes the block (and model) *multirate* since the input and output frame rates will not be equal.
  - **Upsample factor** = `4`
  - **Sample offset** = `0`
  - **Initial condition** = `-1`
  - **Frame-based mode** = **Maintain input frame size**
- Set the **Sample time** parameter of the Digital Clock block to `0.25` to match the sample period of the Upsample block's output.

- Set the **Output signal** parameter of the Frame Status Conversion block to **Sample-based**.

Notice that the current value of the Simulink timer (from the Digital Clock block) is prepended to each output frame. The frame-based signal is converted to a sample-based signal by the Frame Status Conversion block so that the output in the command window will be easily readable.

In the example, the Signal From Workspace block generates a new frame containing four samples once every second (T_{fo} = ¼∗4). The first few output frames are shown below:

(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.

Question: When does the first input sample appear in the output?

The "Latency and Initial Conditions" section of the reference page for the Upsample block 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 therefore predicts that the first input in this example should appear as output sample 17 (i.e., 4∗4+0+1).

To verify this, look at the output from the simulation, saved in the workspace array `yout`. To convert the array to a easier-to-read matrix format, type

```
squeeze(yout)'
```

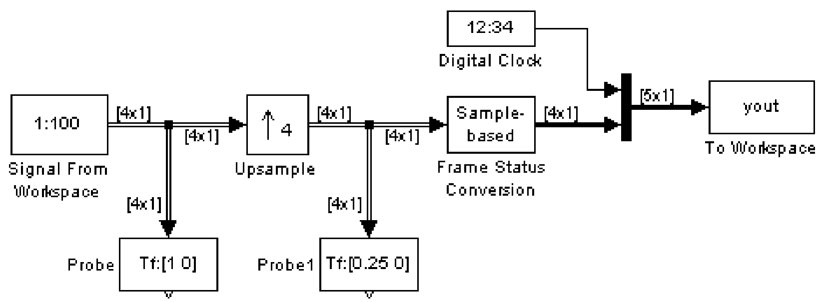
The first few samples of the result, `ans`, are shown below:

```
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					

The first column of yout 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 block's **Initial conditions** parameter. As a result of the tasking latency, the first input value appears only as the first sample of the 5th output frame (at t=1), which is sample 17.

Example: Zero Tasking Latency. Now try the previous example in the Simulink single-tasking mode. The model and all of the block parameter settings are the same.



To engage the Simulink single-tasking mode, adjust the following settings in the **Solver** panel of the **Simulation Parameters** dialog box:

- **Type = Fixed-step**
- **Mode = SingleTasking**

When does the first input sample appear in the output?

The “Latency and Initial Conditions” section of the reference page for Upsample indicates that the block has zero latency for all multirate operations in the Simulink single-tasking mode. To verify this, look at the output from the simulation, `squeeze(yout)'`. The first few samples are shown below:

```
ans =

          0    1.0000         0         0         0   1st output frame
     0.2500    2.0000         0         0         0
     0.5000    3.0000         0         0         0
     0.7500    4.0000         0         0         0
     1.0000    5.0000         0         0         0   5th output frame
     1.2500    6.0000         0         0         0
     1.5000    7.0000         0         0         0
     1.7500    8.0000         0         0         0
     2.0000    9.0000         0         0         0
      time
```

The first column of `yout` 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 input value appears as the first sample of the first output frame (at t=0), as expected for zero-latency operation. Running this model under the Simulink single-tasking mode therefore eliminates the 17-sample delay that the model experiences under the Simulink multitasking mode (for the particular parameter settings in the example).

4

Filters

The DSP Blockset Filtering library provides an extensive array of filtering blocks for analyzing, designing, and implementing filters in your models. If you prefer, you can design and analyze your filters using the Signal Processing Toolbox and Filter Design Toolbox, and then implement the filters in your model using DSP Blockset filter implementation blocks.

Three Main Filter Blocks. There are three main filtering blocks that should satisfy most of your filtering needs:

- Digital Filter Design
  - For designing and analyzing filters, and then realizing the filters you designed.
  - Filters singlechannel and multichannel signals.
  - Simulates single- and double-precision floating-point filters
  - Generates highly optimized C code suitable for use in embedded systems.
- Digital Filter
  - For realizing predesigned filters, that is, filters for which you already have coefficients.
  - Filters singlechannel and multichannel signals.
  - Simulate single- and double-precision floating-point filters
  - Generates highly optimized C code suitable for use in embedded systems.
- Filter Realization Wizard
  - For implementing fixed-point filters using Sum, Gain, and Unit Delay blocks.
  - You can either design and analyze the filter within the Filter Realization Wizard, or you can skip the design and analysis stage if you have a predesigned filter.
  - Filters singlechannel signals
  - Numerically models fixed-point filters in DSP chips, FPGAs, and ASICs. Filters single-channel signals.

All other filter blocks design or implement specialized filters such as frequency-domain filters, filter banks, and adaptive filters.

C Code Generated from Filter Blocks. The Real-Time Workshop C code generated from key filtering blocks is highly optimized for both speed and memory use, and is often suitable for use in embedded systems. For more information on generating C code from models, see Appendix B, “Code Generation Support.”

Topics Covered. The following topics summarize the capabilities of the filter blocks, and provide examples of how to use the key filter blocks:

- “Designing, Analyzing, and Implementing Filters” on page 4-4
- “Implementing Predesigned Filters” on page 4-22
- “Multirate Filters” on page 4-32
- “Adaptive Filters” on page 4-34
- “Analog IIR Filters” on page 4-35

Note You can open many of the example models in the online version of this document by clicking indicated links or by typing specified commands at the MATLAB command line. The links for opening example models do not work in Web browsers; they work only in the MATLAB Help browser, which you can open by typing `doc` at the MATLAB command line.

Designing, Analyzing, and Implementing Filters

The DSP Blockset Filter Designs library offers the Digital Filter Design block and the Filter Realization Wizard for designing, analyzing, and then implementing digital filters. Both provide the same filter design and analysis tools, but use different methods of filter implementation suited for different needs. You can use either block to filter signals in your simulations.

Alternatively, you can use other MathWorks products to design and analyze your filters, and then implement them in your models using filter implementation blocks.

Topics Covered. The following topics show you how to design, analyze, and then implement filters:

- "Choosing Between Digital Filter Design Block and Filter Realization Wizard" on page 4-4 — Guidelines for choosing which block to use to design, analyze, and implement your filter
- "Filter Design, Analysis, and Implementation with the Digital Filter Design Block" on page 4-6 — Example of how to use the Digital Filter Design block
- "Filter Analysis and Design with Other MathWorks Products" on page 4-21 — Summary of how to use Signal Processing Toolbox and Filter Design Toolbox to design and analyze filters.

Related Topics. For a list of other DSP Blockset filtering topics such as multirate filtering or implementing predesigned standard digital filters, see "Topics Covered" on page 4-3 in the overview of the topic on filtering.

Choosing Between Digital Filter Design Block and Filter Realization Wizard

The Digital Filter Design block and Filter Realization Wizard both let you design, analyze, and then implement a filter. To understand which block is best suited for your needs, see the following sections:

- "Similarities" on page 4-5
- "Differences" on page 4-5
- "When to Use Each Block" on page 4-5

Similarities

- **Filter design and analysis options** — Both blocks provide the Filter Design and Analysis Tool (FDATool) GUI for filter design and analysis.
- **Output values** — In double precision, both blocks' outputs numerically match the outputs of the `filter` function in the Filter Design Toolbox and the `filter` function in the Signal Processing Toolbox.

Differences

- **Data type support** — Both blocks support single- and double-precision floating-point computation, but the Filter Realization Wizard additionally supports fixed-point computation.
- **Filter implementation method**
  - The Digital Filter Design block implements very efficient filters that are optimized for both speed and memory use in simulation and in C code generation. For more information on code generation, see Appendix B, "Code Generation Support".
  - The Filter Realization Wizard implements filters using Sum, Gain, and Unit Delay blocks (from either DSP Blockset or Fixed-Point Blockset). These filters are much less efficient than those implemented by the Digital Filter Design block, but both blocks give numerically equivalent results.
- **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. See the Filter Realization Wizard and Digital Filter Design online 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.

When to Use Each Block

- **Digital Filter Design**
  - Use to simulate single- and double-precision floating-point filters.
  - Use to filter multichannel signals

  - Use to generate highly optimized ANSI C code that implements floating-point filters for embedded systems. For more information on code generation, see Appendix B, “Code Generation Support”.
- **Filter Realization Wizard**
  - Use to simulate numerical behavior of fixed-point filters in a DSP chip, FPGA, or ASIC.
  - Use to simulate single- and double-precision floating-point filters with structures that the Digital Filter Design does not support.
  - Use to visualize the filter structure (it builds the filter from Sum, Gain, and Integer Delay blocks)

Filter Design, Analysis, and Implementation with the Digital Filter Design Block

The Digital Filter Design block (in the Filter Designs library) is useful for designing a digital FIR or IIR filter from scratch, and then implementing it. (To implement a filter you already designed, see “Implementing Predesigned Filters with the Digital Filter Block” on page 4-23.)

The Digital Filter Design block is ideal for simulating the numerical behavior of your filter on a single- or double-precision floating-point system, such as a personal computer or DSP chip. The block also generates highly-optimized Real-Time Workshop C code for use in floating-point embedded systems.

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 plots and impulse response plots.

Filter Implementation. The block automatically realizes the filter you design using the filter structure you specify, so you can use the block to filter signals in your models. You can also change some of the filter specification parameters while the block runs in a simulation, which may be useful for fine-tuning the filter. 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 the Signal Processing Toolbox.

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.

Examples and Related Topics. See the following examples and documentation to learn how to use the Digital Filter Design block:

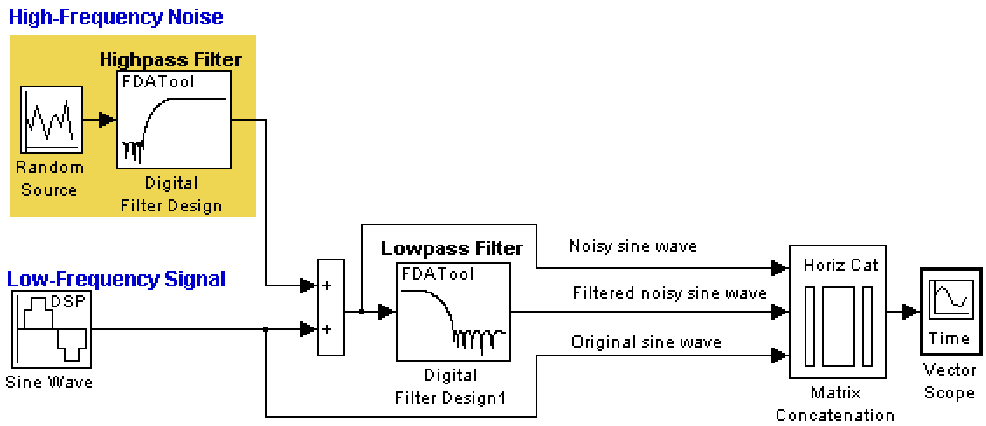
- "Example: Using the Digital Filter Design Block to Design, Analyze, and Implement a Filter" on page 4-8
- "Example: Saving, Importing, and Exporting Filters" on page 4-18
- Digital Filter Design block online reference page — Details about the capabilities of the Digital Filter Design block
- The Filter Design and Analysis Tool (FDATool) GUI topic in the Signal Processing Toolbox documentation — All about the GUI you use to design filters with the Digital Filter Design block.
- "Topics Covered" on page 4-4 in the section about designing and then implementing filters — List of other DSP Blockset filtering topics related to designing, analyzing, and implementing filters.

Example: Using the Digital Filter Design Block to Design, Analyze, and Implement a Filter

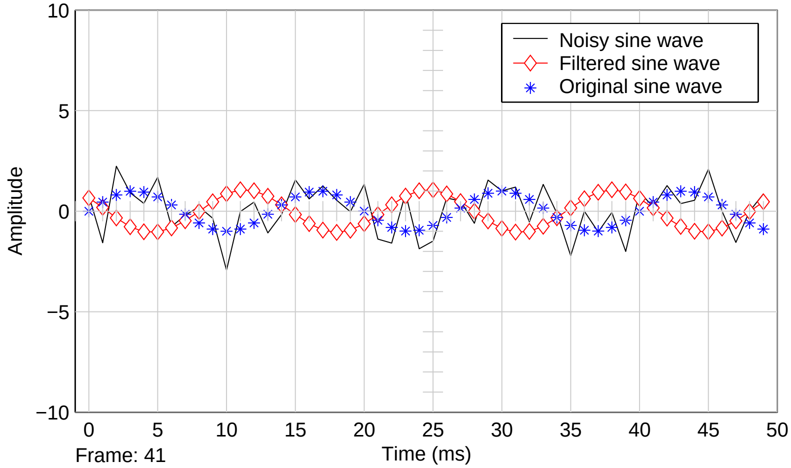
In the following model, a lowpass filter filters out high-frequency noise from a noisy sine wave. The high-frequency noise is output by a highpass filter excited by a uniform random signal.

Digital Filter Design blocks were used to design and implement the lowpass and highpass filters. The Vector Scope's display shows the original sine wave, the noisy sine wave, and the filtered noisy sine wave for comparison.

Note You can open the following model by clicking here in the MATLAB Help browser (*not* in a Web browser). To build the model yourself, follow the steps below. For a review of how to create and run Simulink models, see Chapter 2, "Simulink Review." For detailed information on the Digital Filter Design block, see the Digital Filter Design block online reference page.



Model Using the Digital Filter Design Block to Implement Filters



Vector Scope Display After Running the Model

To build the model yourself, complete the following steps:

- "Step 1 — Get Necessary Blocks" on page 4-10
- "Step 2 — Design the Lowpass Filter and Select a Filter Structure" on page 4-10
- "Step 3 — Design the Highpass Filter and Select a Filter Structure" on page 4-12
- "Step 4 — Set the Rest of the Blocks' Parameters and Connect the Blocks" on page 4-14
- "Step 5 — Set Simulation Parameters and Run the Model" on page 4-15
- "Step 6 — Set the Vector Scope Display Colors" on page 4-16
- "Step 7 — Change the Filter While the Simulation Runs" on page 4-17

Step 1 — Get Necessary Blocks. Place the blocks listed in the following table into a new model.

Block	Library	Quantity
Digital Filter Design	Filter Designs (sublibrary of Filters library)	2
Matrix Concatenation	Matrix Operations (sublibrary of Matrices and Linear Algebra library, which is a sublibrary of the Math Functions library)	1
Random Source	DSP Sources	1
Sine Wave	DSP Sources	1
Sum	The Simulink Math Operations library	1
Vector Scope	DSP Sinks	1

Step 2 — Design the Lowpass Filter and Select a Filter Structure.

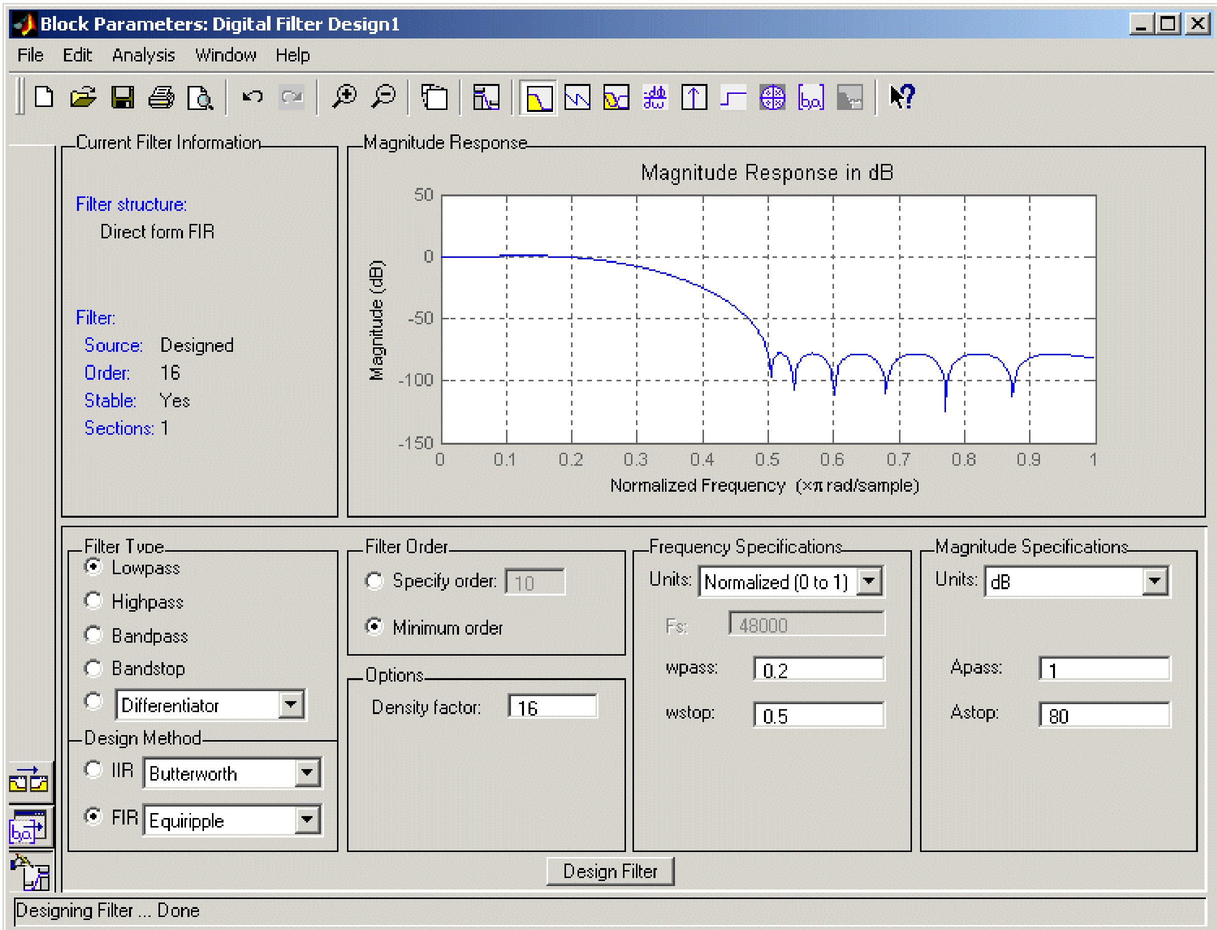
a Double-click one of the Digital Filter Design blocks in the model to open the Filter Design and Analysis Tool (FDATool) GUI, shown in the following figure.

b Set the block parameters as indicated in the following figure, and click **Design Filter** at the bottom of the GUI to design the filter.

c Click the **Magnitude Response** button  to view the magnitude response of the filter as in the following figure. (You can explore the other buttons, which provide other filter analysis tools such as pole-zero plots and impulse response plots. Clicking the **Filter Specification** button  shows you the original display.)

d Set the filter's structure in the **Edit** menu by selecting **Convert Structure...**. Set the structure to **Direct form FIR transpose**.

The block now implements a lowpass filter with a direct form FIR transpose structure. As the **wpass** and **wstop** settings indicate, 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.


Block Parameters: Digital Filter Design1
File Edit Analysis Window Help
Current Filter Information
Filter structure:
Direct form FIR
Filter:
Source: Designed
Order: 16
Stable: Yes
Sections: 1
Magnitude Response
Magnitude Response in dB
Magnitude (dB)
50
0
-50
-100
-150
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1
Normalized Frequency (×π rad/sample)
Filter Type
Lowpass
Highpass
Bandpass
Bandstop
Differentiator
Design Method
IIR Butterworth
FIR Equiripple
Filter Order
Specify order: 10
Minimum order
Options
Density factor: 16
Frequency Specifications
Units: Normalized (0 to 1)
Fs: 48000
wpass: 0.2
wstop: 0.5
Magnitude Specifications
Units: dB
Apass: 1
Astop: 80
Design Filter
Designing Filter ... Done

Step 3 — Design the Highpass Filter and Select a Filter Structure. (This step is similar to what you did in the "Step 2 — Design the Lowpass Filter and Select a Filter Structure" section.)

a Place a second Digital Filter Design block from the DSP Blockset Filter Designs library into your model.

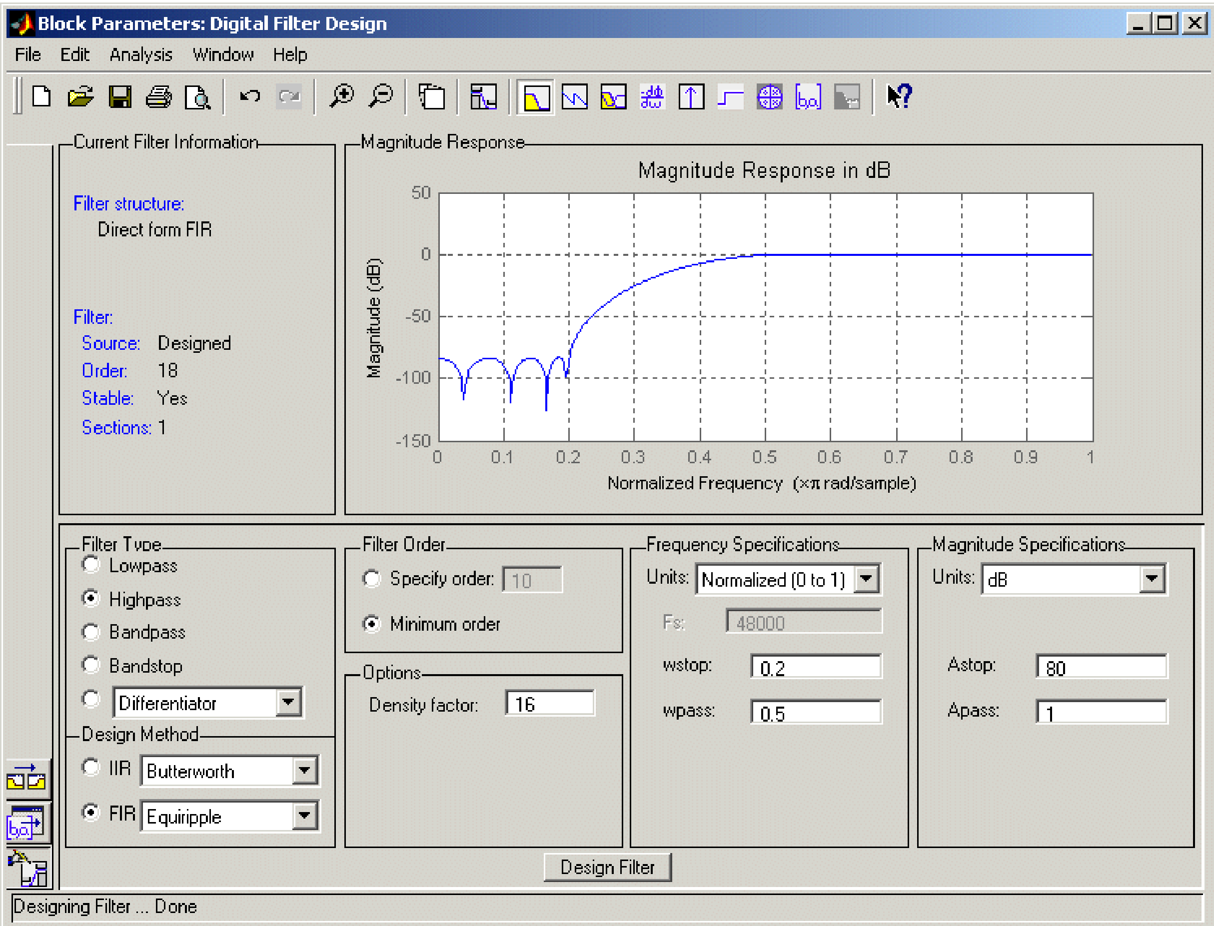
b Set this block's parameters as shown in the following figure.

c Set the filter structure to direct form FIR transpose (in the **Edit** menu by selecting **Convert Structure...**).

d Click the **Magnitude Response** button  to view the magnitude response of the filter. (To see the original display, click the **Filter Specification** button .)

The block now implements a highpass filter with a direct form FIR transpose structure. As the **wpass** and **wstop** settings indicate, 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.

Note that this highpass filter is the "opposite" of the lowpass filter you just designed, in that this filter passes the frequencies stopped by the lowpass filter, and stops the frequencies passed by the lowpass filter. (So, in the model you are building, the lowpass filter should be able to filter out the high-frequency noise output by this highpass filter.)


Block Parameters: Digital Filter Design
File Edit Analysis Window Help
Current Filter Information
Filter structure:
Direct form FIR
Filter:
Source: Designed
Order: 18
Stable: Yes
Sections: 1
Magnitude Response
Magnitude Response in dB
Magnitude (dB)
50
0
-50
-100
-150
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
Normalized Frequency (×π rad/sample)
Filter Type
Lowpass
Highpass
Bandpass
Bandstop
Differentiator
Design Method
IIR
Butterworth
FIR
Equiripple
Filter Order
Specify order:
10
Minimum order
Options
Density factor:
16
Frequency Specifications
Units:
Normalized (0 to 1)
Fs:
48000
wstop:
0.2
wpass:
0.5
Magnitude Specifications
Units:
dB
Astop:
80
Apass:
1
Design Filter
Designing Filter ... Done

Step 4 — Set the Rest of the Blocks' Parameters and Connect the Blocks.

a. Set the parameters for the rest of the blocks as indicated in the following table; leave any parameters not listed in the table in their default settings.

b. Connect the blocks as shown in Model Using the Digital Filter Design Block to Implement Filters on page 4-8. (You may need to resize some of the blocks to make your model look like the figure.)

Parameter Settings for the Other Blocks

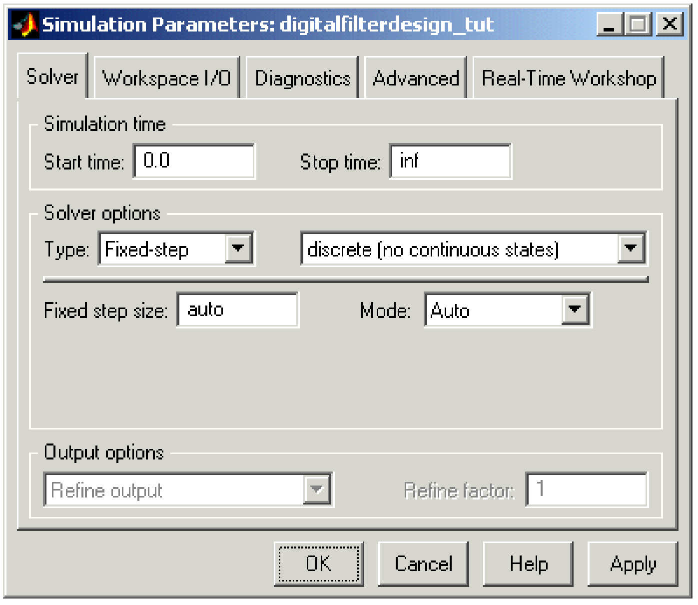
Block	Parameter Setting
Matrix Concatenation	• **Number of inputs** — 3 • **Concatenation method** — **Horizontal**
Random Source	• **Source type** — **Uniform** • **Minimum** — 0 • **Maximum** — 4 • **Sample mode** — **Discrete** • **Sample time** — 1/1000 • **Samples per frame** — 50
Sine Wave	• **Frequency (Hz)** — 75 • **Sample time** — 1/1000 • **Samples per frame** — 50
Sum	• **Icon shape** — **rectangular** • **List of signs** — ++
Vector Scope	**Scope properties:** • **Input domain** — **Time** • **Time display span (number of frames)** — 1

Step 5 — Set Simulation Parameters and Run the Model.

a Open the **Simulation Parameters** dialog box (click the **Simulation** menu in your model and select **Simulation parameters...**).

b Set the **Simulation Parameters** dialog box settings as indicated in the following diagram.

c Run the simulation (go to the **Simulation** menu in your model and select **Start**). When you finish observing the running model, stop the simulation (select **Stop** from the **Simulation** menu).



Appropriate Simulation Parameters Settings

Step 6 — Set the Vector Scope Display Colors. The Vector Scope's display can show the three signals (noisy sine wave, filtered noisy sine wave, and original sine wave) using different colors and line styles. It can also show a channel legend:

a Show the channel legend by clicking the **Axes** menu and selecting **Channel legend**.

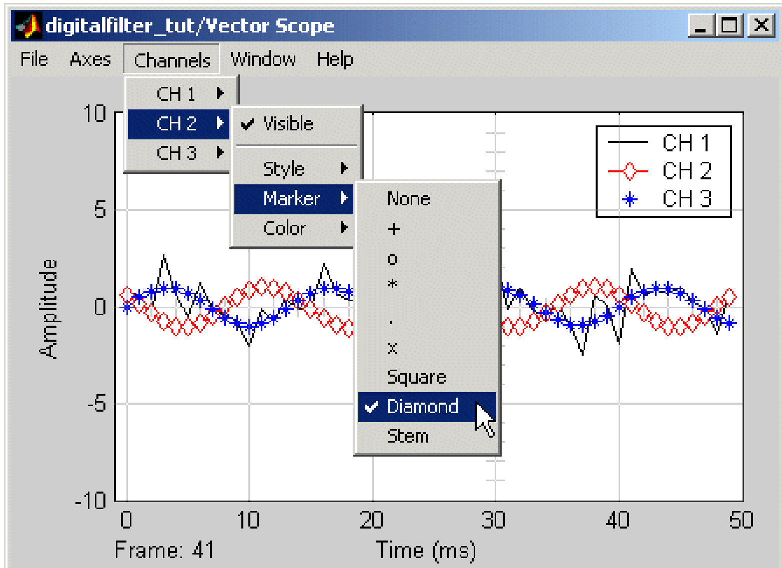
b See the options for setting the color, style, and marker of each channel by clicking the **Channels** menu in the Vector Scope's display. If you connected your blocks as in Model Using the Digital Filter Design Block to Implement Filters on page 4-8, the channels in the Vector Scope display (listed in the **Channels** menu) correspond to the following signals:

Channel 1 — Noisy sine wave

Channel 2 — Filtered noisy sine wave

Channel 3 — Original sine wave

c 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.



Setting Vector Scope Display Colors and Markers

Step 7 — Change the Filter While the Simulation Runs. You can change your filter parameters while the simulation runs as long as your changes do not alter the filter length or filter order; this is useful for fine-tuning your filter designs:

- **a** Run the simulation (go to the **Simulation** menu and select **Start**).
- **b** Double-click the Digital Filter Design block that implements the lowpass filter.
- **c** Click the **Magnitude Response** button  to view the magnitude response of the filter.
- **d** Change **wpass** to `0.4` and **wstop** to `0.6`; these changes should cause the filter to filter out less of the noise than it did before.
- **e** Click **Design Filter** at the bottom of the GUI. Notice the magnitude response plot updates in the block, as does the block's icon in the model.
- **f** When you finish observing the running model, stop the simulation (select **Stop** from the **Simulation** menu). Note that the Vector Scope display shows that the lowpass filter no longer does as good a job of filtering the noise from the noisy sine wave.

Example: Saving, Importing, and Exporting Filters

Double-clicking the Digital Filter Design block or the Filter Realization Wizard opens the Filter Design and Analysis Tool (FDATool) GUI. FDATool allows you to save, import, and export filters you design. You can use these features as follows:

- Saving a filter design session allows you to save the current filter design settings in FDATool so you can open it later to continue your filter design work.
- Importing filters designed elsewhere allows you to redesign or analyze the filter with FDATool.
- Exporting a filter to the MATLAB workspace or a MAT-file allows other DSP Blockset filter blocks and other applications to use the filter.

The following topics show you how to use FDATool's saving, importing, and exporting features, which apply to both the Digital Filter Design block and Filter Realization Wizard:

- "Saving an FDATool Filter Design Session" on page 4-18
- "Importing a Filter to FDATool" on page 4-19
- "Exporting a Filter from FDATool" on page 4-20
- Signal Processing Toolbox documentation on FDATool

Saving an FDATool Filter Design Session. Place a Digital Filter Design block or the Filter Realization Wizard from the Filter Designs library in a model. Double-click the block to open FDATool, which opens with a default filter already designed. (To view its frequency response, click the **Magnitude Response** button .) Try changing the filter design (for instance, by changing the filter to a highpass filter).

You can save the current filter design settings in FDATool by saving the filter design session as follows:

a Click the **File** menu and select **Save Session As...**.

b Save the session in a directory of your choice in the **Save Filter Design Session** dialog box.

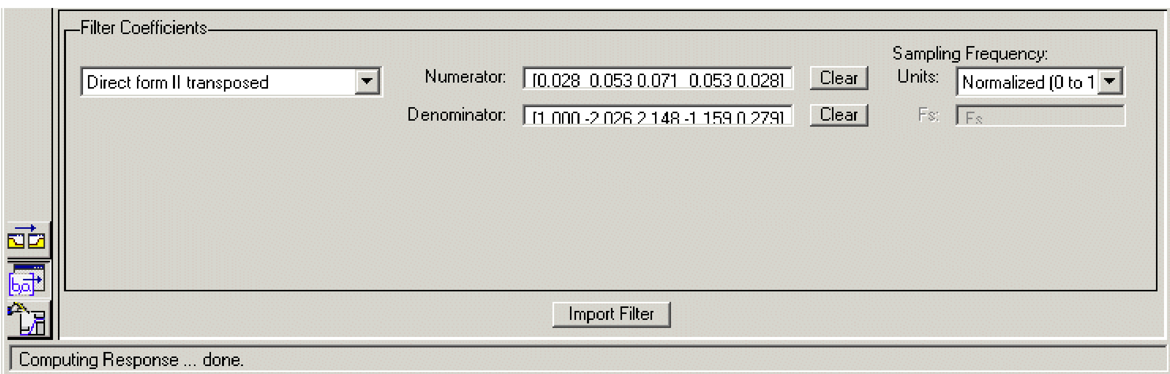
Next time you open FDATool, you can load this saved session by clicking the **File** menu and selecting **Open Session...**.

Importing a Filter to FDATool. Place a Digital Filter Design block or the Filter Realization Wizard from the Filter Designs library in a model. Double-click the block to open FDATool. You can import a filter designed outside of FDATool (for instance, at the MATLAB command line using Signal Processing Toolbox or Filter Design Toolbox) as follows:

a Click the **Import Filter** button at the lower left-hand side of FDATool

b Set the parameters in the import filter panel (shown in The Import Filter Panel in FDATool on page 4-19). Note that you can type in a vector of filter coefficients directly or type in the name of a MATLAB workspace variable containing the coefficients.

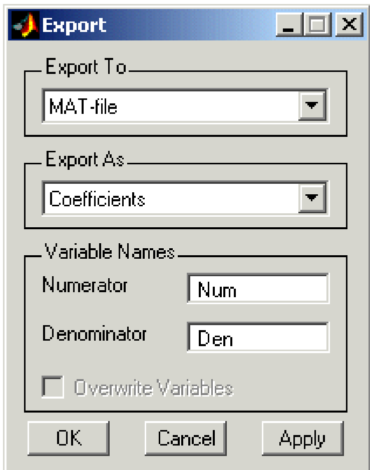
c Click the **Filter Design** button at the lower left-hand side of FDATool to view the filter design panel. Now you can redesign or analyze the imported filter with FDATool.



The Import Filter Panel in FDATool

Exporting a Filter from FDATool. Place a Digital Filter Design block or the Filter Realization Wizard from the Filter Designs library in a model. Double-click the block to open FDATool. You can export a filter you designed in FDATool to the MATLAB workspace, MAT-file, or text file as follows:

a Click the **File** menu and select **Export Filter...**.



b Set the parameters in the **Export** dialog. You can export to the MATLAB workspace, to a MAT-file, or to a text file. To use the filter in another filter block, export to the workspace or to a MAT-file, which requires you to type in the names of the variables in which to store the filter coefficients.

c Click **OK** to export the filter. Now you can use the variables in other filter blocks and applications.

Filter Analysis and Design with Other MathWorks Products

In addition to using DSP Blockset blocks for designing filters, you can also use the functions and GUIs provided by the Signal Processing Toolbox and Filter Design Toolbox to design and analyze your filters.

Once you design a filter using either toolbox, you can use the DSP Blockset's filter implementation blocks to realize the filters in your models in one of the following ways:

- Cut and paste the filter coefficients into the filter coefficient parameters in a filter implementation block.
- Store the filter coefficients in a variable in the MATLAB workspace or a MAT-file, and type that variable name into appropriate parameters of a filter implementation block.
- Type expressions that compute the filter coefficients into appropriate parameters of a filter implementation block.

For a list of topics about the filter implementation blocks, see the DSP Blockset Filtering library.

Implementing Predesigned Filters

The DSP Blockset Filter Designs library provides several blocks that implement digital FIR and IIR filters in your models. Some of these blocks allow you to first design and analyze a filter, and then implement the filter, as described in "Designing, Analyzing, and Implementing Filters" on page 4-4. This section shows you how to use the blocks that let you skip the design and analysis step, and let you provide the filter coefficients directly.

The following are the primary blocks for implementing filters for which you already know the coefficients:

- The Digital Filter block supports many filter structures, and can filter multichannel signals. This block generates highly optimized Real-Time Workshop C code that is suitable for use in floating-point embedded systems.
- The Filter Realization Wizard supports many filter structures, and realizes filters using sum, gain, and delay blocks so you can easily visualize your filter's structure. Use this block to simulate the *numerical* behavior of *fixed-point filters* on DSPs, FPGAs, and ASICs. (This block allows you to either design your filter from scratch or directly provide your filter's coefficients.)

Note The DSP Blockset also provides blocks for implementing frequency-domain filters, time-varying filters, multirate filters (such as filter banks and filters for decimation and interpolation), and adaptive filters. For more information, see "Topics Covered" on page 4-3.

Examples and Other Related Topics. The following topics have information related to implementing standard digital filters using the DSP Blockset:

- "Example: Using the Digital Filter Block to Implement a Predesigned Filter" on page 4-24 — How to implement filters using the Digital Filter block.
- Online references of the Digital Filter block — Detailed information about the filter implementation capabilities of the Digital Filter block.
- "Designing, Analyzing, and Implementing Filters" on page 4-4 — How to design and analyze filters using DSP Blockset and other MathWorks products

- “Multirate Filters” on page 4-32 — Blocks for implementing multirate filters such as filter banks and FIR filters for decimation and interpolation
- “Adaptive Filters” on page 4-34 — Blocks for implementing adaptive filters such as Kalman and LMS adaptive filters
- “Analog IIR Filters” on page 4-35 — Using the block for designing and implement analog filters in the DSP Blockset.

Implementing Predesigned Filters with the Digital Filter Block

The Digital Filter block in the Filter Designs library is useful for implementing a predesigned filter whose coefficients you already know. (If you want to first design a filter and then implement it, see “Filter Design, Analysis, and Implementation with the Digital Filter Design Block” on page 4-6.) The block is ideal for simulating the numerical behavior of your filter on a single- or double-precision floating-point system, such as a personal computer or DSP chip. The block also generates highly-optimized Real-Time Workshop C code for use in floating-point embedded systems.

Required Filter Parameters for Using the Digital Filter Block. To implement a filter with the Digital Filter block, you must provide the following basic information about the filter:

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

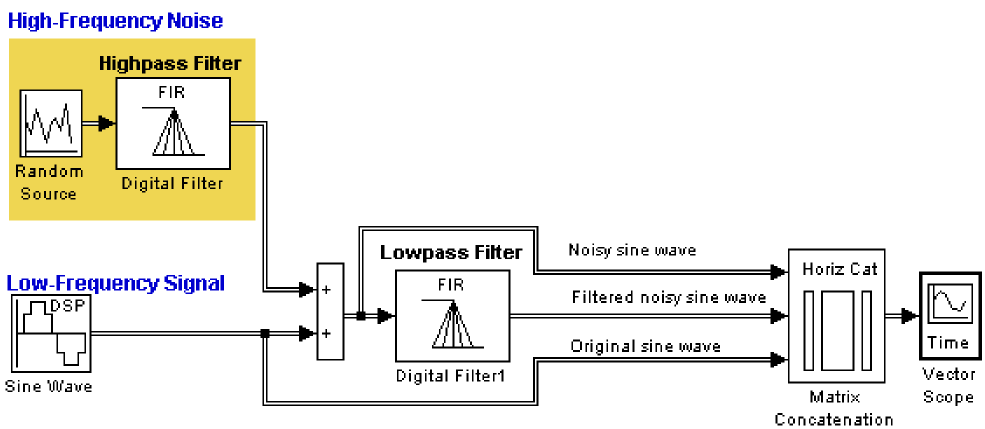
Related Topics. For more information, see “Examples and Other Related Topics” on page 4-22 in the topic on implementing predesigned filters.

Example: Using the Digital Filter Block to Implement a Predesigned Filter

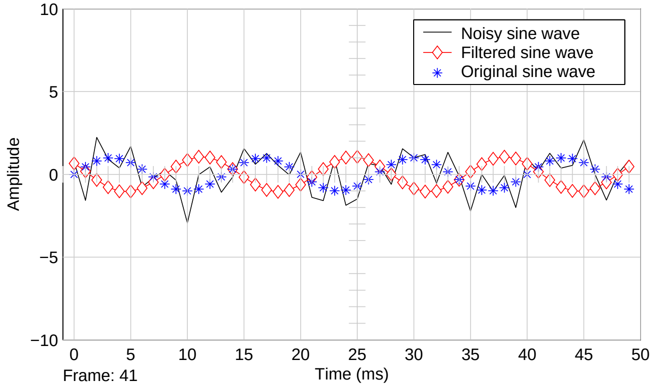
In the following model, a lowpass filter filters out high-frequency noise from a noisy sine wave. The high-frequency noise is output by a highpass filter excited by a uniform random signal.

Digital Filter blocks implement both the lowpass and highpass filters, which were predesigned elsewhere (perhaps at the MATLAB command line, or in a Digital Filter Design block). The Vector Scope's display (see Vector Scope Display After Running the Model on page 4-25) shows the original sine wave, the noisy sine wave, and the filtered noisy sine wave for comparison.

Note You can open the following model by clicking here in the MATLAB Help browser (*not* in a Web browser). To build the model yourself, follow the steps below. For a review of how to create and run Simulink models, see Chapter 2, "Simulink Review." For detailed information on the Digital Filter block, see the Digital Filter block online reference page.



Model Using the Digital Filter Block to Implement Filters



Vector Scope Display After Running the Model

To build the model yourself, complete the following steps:

- "Step 1 — Get the Coefficients of the Predesigned Filters" on page 4-26
- "Step 2 — Get Necessary Blocks" on page 4-27
- "Step 3 — Set the Lowpass Digital Filter Block Parameters" on page 4-27
- "Step 4 — Set the Highpass Digital Filter Block Parameters" on page 4-28
- "Step 5 — Set the Rest of the Blocks' Parameters and Connect the Blocks" on page 4-29
- "Step 6 — Set Simulation Parameters and Run the Model" on page 4-30
- "Step 7 — Set the Vector Scope Display Colors" on page 4-31

Note that the above model is almost identical to the example model in "Example: Using the Digital Filter Design Block to Design, Analyze, and Implement a Filter" on page 4-8. The only difference is that in this model, Digital Filter blocks implement the filters (rather than Digital Filter Design blocks). Given the same filter design, both blocks implement the filter identically and behave the same in both simulation and code generation. The difference between the blocks is not their filter implementation capabilities, but the manner in which you specify the filter to implement: in the Digital Filter block, you must type in the filter coefficients, whereas in the Digital Filter Design block, you design the filter using a GUI.

Step 1 — Get the Coefficients of the Predesigned Filters. For the purposes of this example, we assume you already predesigned your lowpass and highpass filters, possibly by using one of the following filter design techniques:

- **Use Signal Processing Toolbox functions** — Type the following commands at the MATLAB command line:

```
[N Fo Ao W] = remezord([0.2 0.5],[1 0], ...
[5.750112778453722e-002,1.000000000000001e-004]);

lopassNum = remez(N,Fo,Ao,W,{16});

[N Fo Ao W] = remezord([0.2 0.5],[0 1], ...
[1.000000000000001e-004 5.750112778453722e-002]);

hipassNum = remez(N,Fo,Ao,W,{16});
```

- **Use Digital Filter Design block** — Rather than type the above commands, use the Digital Filter Design block to design the same filters as in “Example: Using the Digital Filter Design Block to Design, Analyze, and Implement a Filter” on page 4-8. Then export the filters to the MATLAB workspace to the variables `lopassNum` and `hipassNum` as described in “Exporting a Filter from FDATool” on page 4-20.

Step 2 — Get Necessary Blocks. Place the blocks in the following table into a new model.

Block	Library	Quantity
Digital Filter	Filter Designs (sublibrary of Filters library)	2
Matrix Concatenation	Matrix Operations (sublibrary of Matrices and Linear Algebra library, which is a sublibrary of the Math Functions library)	1
Random Source	DSP Sources	1
Sine Wave	DSP Sources	1
Sum	The Simulink Math library	1
Vector Scope	DSP Sinks	1

Step 3 — Set the Lowpass Digital Filter Block Parameters. Double-click one of the Digital Filter blocks and set its filter parameters as indicated in Figure ; this block is the lowpass filter that filters out the noise from the noisy sine wave signal.

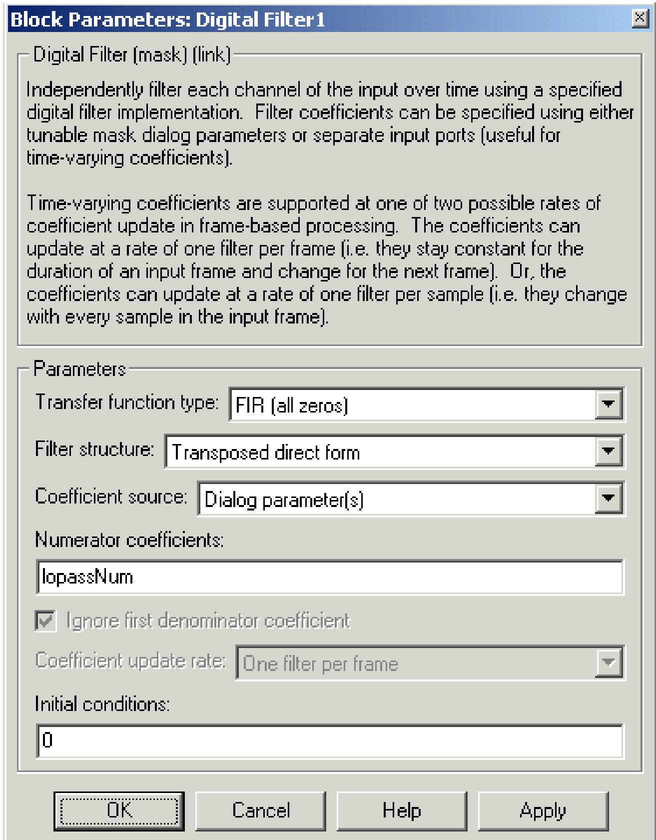
Note that when implementing your lowpass filter (and any other filter) with the Digital Filter block, you need to provide the following basic information about your filter:

- 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

Also note that you can provide the filter coefficients in several ways:

- Type in a variable name from the MATLAB workspace (such as `lopassNum` in Figure). The variable could be a result of running filter design commands or of using the Digital Filter Design block to design a filter.
- 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. If your vector of filter coefficients is large, rather than typing in the vector you can assign it to a variable in the MATLAB workspace and type in the variable name instead.



Lowpass Digital Filter Block Settings

Step 4 — Set the Highpass Digital Filter Block Parameters. Set the other Digital Filter block with the same settings as shown in Figure , except change the **Numerator coefficients** parameter to `hipassNum`; this block implements the highpass filter that outputs the high-frequency noise.

Step 5 — Set the Rest of the Blocks' Parameters and Connect the Blocks.

a Set the parameters for the rest of the blocks as indicated in the following table; leave any parameters not listed in the table in their default settings.

b Connect the blocks as shown in Model Using the Digital Filter Block to Implement Filters on page 4-24. (You may need to resize some of the blocks to make your model look like the one in the figure.)

Parameter Settings for the Other Blocks

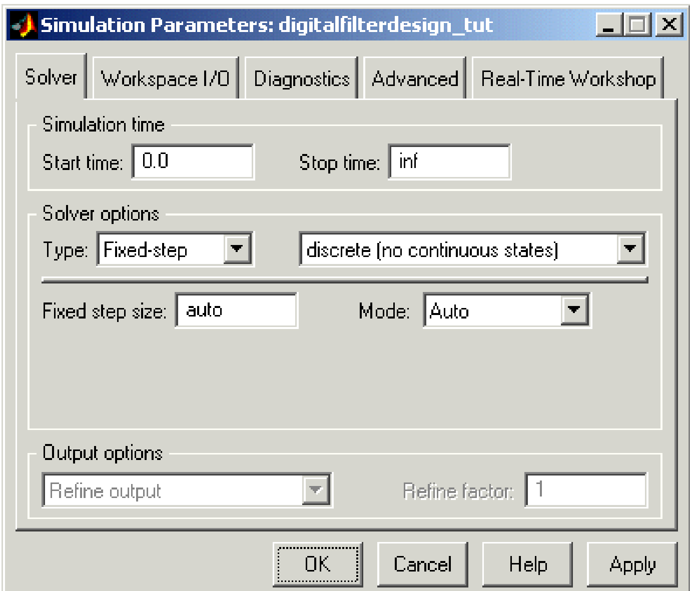
Block	Parameter Setting
Matrix Concatenation	• **Number of inputs** — 3 • **Concatenation method** — **Horizontal**
Random Source	• **Source type** — **Uniform** • **Minimum** — 0 • **Maximum** — 4 • **Sample mode** — **Discrete** • **Sample time** — 1/1000 • **Samples per frame** — 50
Sine Wave	• **Frequency (Hz)** — 75 • **Sample time** — 1/1000 • **Samples per frame** — 50
Sum	• **Icon shape** — **rectangular** • **List of signs** — ++
Vector Scope	**Scope properties:** • **Input domain** — **Time** • **Time display span (number of frames)** — 1

Step 6 — Set Simulation Parameters and Run the Model.

a Open the **Simulation Parameters** dialog box (click the **Simulation** menu in your model and select **Simulation parameters...**).

b Set the **Simulation Parameters** dialog box settings as indicated in the following diagram.

c Run the simulation (go to the **Simulation** menu in your model and select **Start**). When you finish observing the running model, stop the simulation (select **Stop** from the **Simulation** menu).



Appropriate Simulation Parameters Settings

Step 7 — Set the Vector Scope Display Colors. The Vector Scope's display can show the three signals (noisy sine wave, filtered noisy sine wave, and original sine wave) using different colors and line styles. It can also show a channel legend:

a Show the channel legend by clicking the **Axes** menu and selecting **Channel legend**.

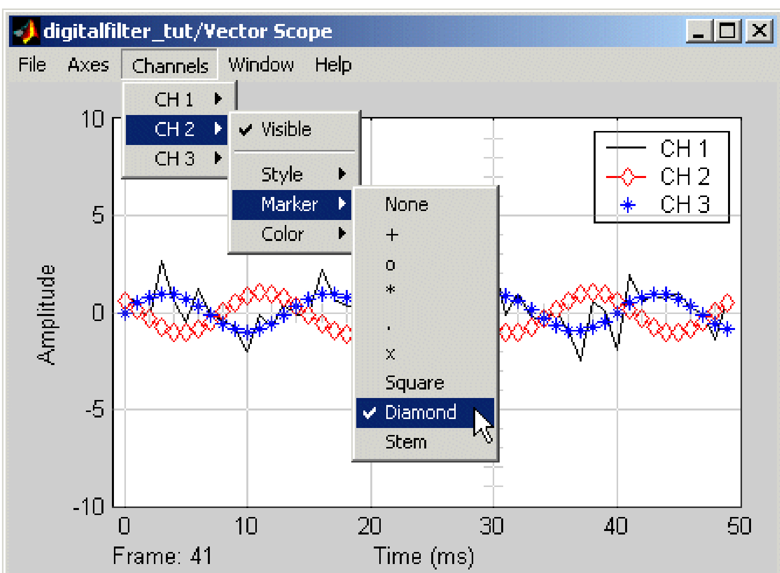
b See the options for setting the color, style, and marker of each channel by clicking the **Channels** menu in the Vector Scope's display. If you connected your blocks as in Model Using the Digital Filter Design Block to Implement Filters on page 4-8, the channels in the Vector Scope display (listed in the **Channels** menu) correspond to the following signals:

Channel 1 — Noisy sine wave

Channel 2 — Filtered noisy sine wave

Channel 3 — Original sine wave

c 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.



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.

The Multirate Filters library provides a number of blocks for multirate applications:

- Dyadic Analysis Filter Bank
- Dyadic Synthesis Filter Bank
- FIR Decimation
- FIR Interpolation
- FIR Rate Conversion
- Two-Channel Analysis Subband Filter
- Two-Channel Synthesis Subband Filter

Multirate Filtering Demos

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

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 DSP Blockset demos, type `demo blockset dsp` at the MATLAB command line.

Multirate Filtering Demos	Commands for Opening Demos in MATLAB
Denoising	`dspwdnois`
Multistage Multirate Filtering Suite	`dspmrf_menu`
Interpolation of a Sinusoidal Signal	`dspintrp`
Sample Rate Conversion	`dspsrcnv`

Multirate Filtering Demos	Commands for Opening Demos in MATLAB
Sigma-Delta A/D Converter	`dspsdadc`
Three-Channel Wavelet Transmultiplexer	`dspwvtrnsmx`
Wavelet Perfect Reconstruction Filter Bank	`dspwpr`
Wavelet Reconstruction	`dspwlet`

Adaptive Filters

Adaptive filters are filters whose transfer function coefficients or *taps* change over time in response to an external error signal. The Adaptive Filters library contains the following blocks:

- Kalman Adaptive Filter
- LMS Adaptive Filter
- RLS Adaptive Filter

Adaptive Filtering Demos

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

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 DSP Blockset demos, type `demo blockset dsp` at the MATLAB command line.

Adaptive Filtering Demos	Commands for Opening Demos in MATLAB
LMS Adaptive Equalization	`lmsadeq`
LMS Adaptive Linear Prediction	`lmsadlp`
LMS Adaptive Noise Cancellation	`lmsdemo`
LMS Adaptive Time-Delay Estimation	`lmsadtde`
Nonstationary Channel Estimation	`kalmnsce`
RLS Adaptive Noise Cancellation	`rlsdemo`

Analog IIR Filters

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

where:

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 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 the "Filter Designs" section of the Signal Processing Toolbox documentation.

Note The Analog Filter Design block does not work with the Simulink discrete solver, which is enabled when the **discrete** option is selected in the **Solver** panel of the **Simulation Parameters** dialog box. Select one of the continuous solvers (e.g., **ode4**) instead.

5

Transforms

The Transforms library provides blocks for a number of transforms that are of particular importance in DSP applications:

- Analytic Signal
- Complex Cepstrum
- DCT
- FFT
- IDCT
- IFFT
- Real Cepstrum

First and foremost among these are of course the FFT and IFFT blocks, which respectively implement the fast Fourier transform and its inverse. These blocks and examples of how to use them are discussed further in the next section.

Using the FFT and IFFT Blocks

This section provides the following two example models that use the FFT and IFFT blocks:

- "Example: Using the FFT Block"
- "Example: Using the IFFT Block" on page 5-4

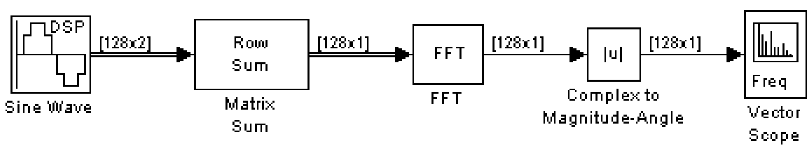
The first example loosely follows the example in the "Discrete Fourier Transform" section of the Signal Processing Toolbox documentation, where you can also find additional background information on these transform operations.

Example: Using the FFT Block

In the model below, the Sine Wave block generates two frame-based sinusoids, one at 15 Hz and the other at 40 Hz. The sinusoids are summed point-by-point to generate the compound sinusoid

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

which is then transformed to the frequency domain using an FFT block.

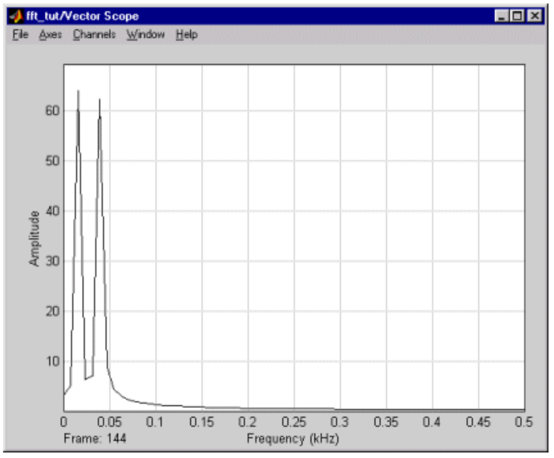


To build the model, make the following parameter settings:

- In the Sine Wave block, set:
  - **Amplitude** = `1`
  - **Frequency** = `[15 40]`
  - **Phase offset** = `0`
  - **Sample time** = `0.001`
  - **Samples per frame** = `128`
- In the Matrix Sum block, set **Sum along** = **Rows**.
- In the Complex to Magnitude-Angle block, set **Output** = **Magnitude**.

- In the Vector Scope block, set:
  - **Input domain** = **Frequency** in the **Scope properties** panel
  - **Amplitude scaling** = **Magnitude** in the **Axis properties** panel
- Set the **Stop time** in the **Parameters** dialog box to `inf`, and start the simulation by selecting **Start** from the **Simulation** menu.

The scope shows the two peaks at 0.015 and 0.04 kHz, as expected.



Note that the three-block sequence of FFT, Complex to Magnitude-Angle, and Vector Scope could be replaced by a single Spectrum Scope block, which computes the magnitude FFT internally.

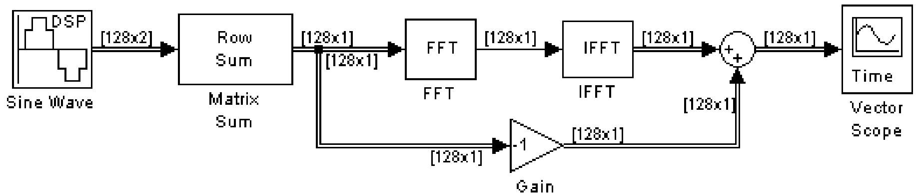
Other blocks that compute the FFT internally are the blocks in the Power Spectrum Estimation library. See "Power Spectrum Estimation" on page 6-6 for more information about these blocks.

Example: Using the IFFT Block

In the model below, the Sine Wave block again generates two frame-based sinusoids, one at 15 Hz and the other at 40 Hz. The sinusoids are summed point-by-point to generate the compound sinusoid

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

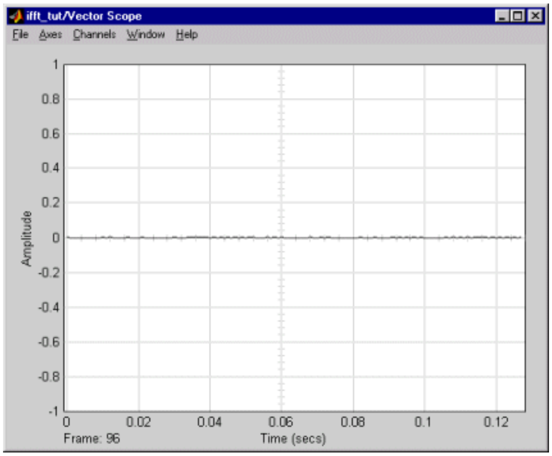
which is transformed to the frequency domain using an FFT block. The frequency-domain signal is then immediately transformed back to the time domain by the IFFT block, and the difference between the original time-domain signal and transformed time-domain signal is plotted on the scope.



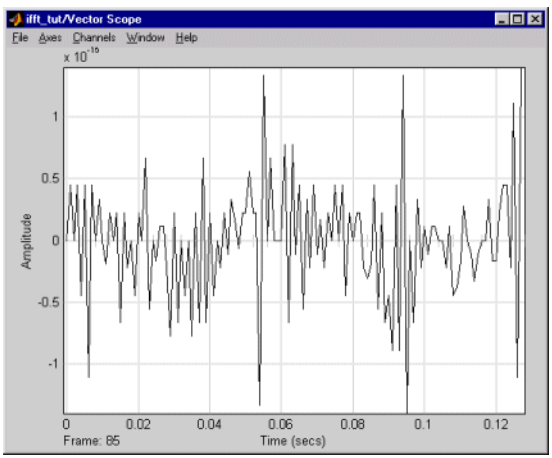
To build the model, make the following parameter settings (leave unlisted parameters in their default settings):

- In the Sine Wave block, set:
  - **Amplitude** = `1`
  - **Frequency** = `[15 40]`
  - **Phase offset** = `0`
  - **Sample time** = `0.001`
  - **Samples per frame** = `128`
- In the Matrix Sum block, set **Sum along** = **Rows**.
- In the FFT block, set **Output in bit-reversed order** = ☑
- In the IFFT block, set:
  - **Input is in bit-reversed order** = ☑
  - **Input is conjugate symmetric** = ☑
- In the Sum block, set **List of signs** = `|++`.
- In the Gain block, set **Gain** = `-1`.
- In the **Scope properties** panel of the Vector Scope block, set **Input domain** = **Time**
- Set the **Stop time** in the **Parameters** dialog box to `inf`, and start the simulation by selecting **Start** from the **Simulation** menu.

The flat line on the scope suggests that there is no difference between the two signals, and that the IFFT block has perfectly reconstructed the original time-domain signal from the frequency-domain input.



More precisely, the two signals are identical to within round-off error, which can be seen by selecting **Autoscale** from the right-click menu on the scope. The enlarged trace shows that the differences between the two signals are on the order of 10^{-15}.



6

Statistics, Estimation, and Linear Algebra

This section covers how you can implement the following basic DSP operations using DSP Blockset:

- "Statistics" on page 6-3
- "Power Spectrum Estimation" on page 6-6
- "Linear Algebra" on page 6-7

The discussion and examples included in these sections should help you become familiar with the standard operations involved in simulating DSP models. See Chapter 3, "Working with Signals" for more basic information on sample rates, matrices, and frame-based processing.

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 unchecked **Running** box means that the block is operating in basic mode, while a checked **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 6-5

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
```

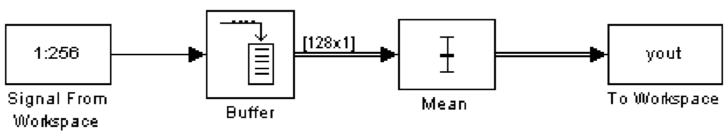
which computes the maximum of each column in input `u`.

The next section provides 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_0) 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_0-L points. The **Buffer overlap** is often M_0-1 (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`

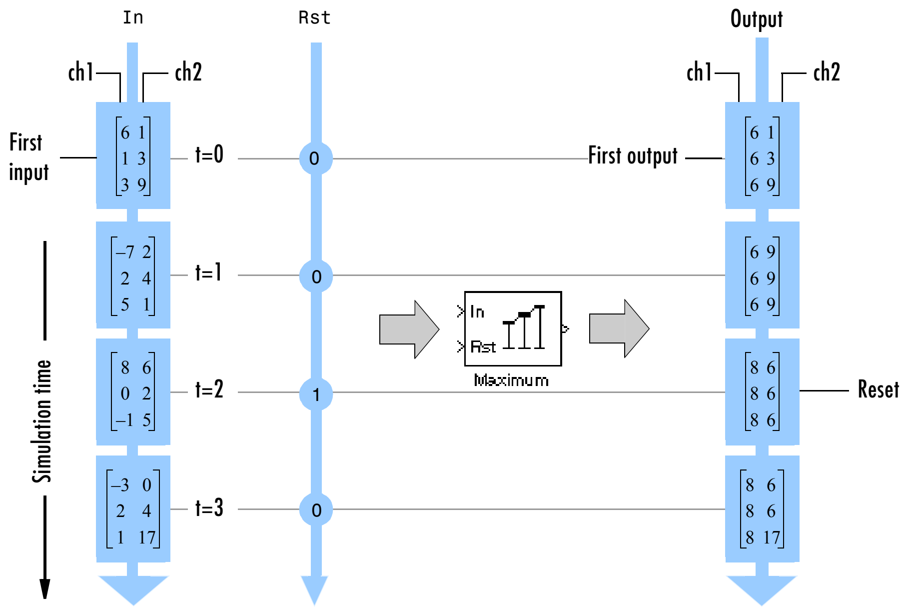
- In the Buffer block, set:
  - **Output buffer size (per channel)** = 128
  - **Buffer overlap** = 127

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 DSP 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 “Multichannel Signals” on page 3-11 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 6-8
- “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

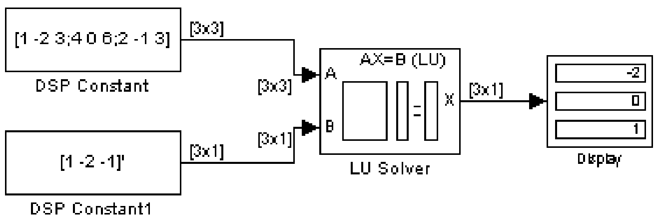
Some of the blocks offer particular strengths for certain classes of problems. For example, the Cholesky Solver block is particularly adapted for a square Hermitian positive definite matrix A, whereas the Backward Substitution block is particularly suited for an upper triangular matrix A.

Example: LU Solver

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

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

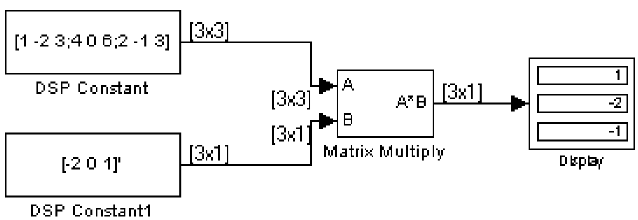
and finds x to be the vector `[-2 0 1]'`.



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

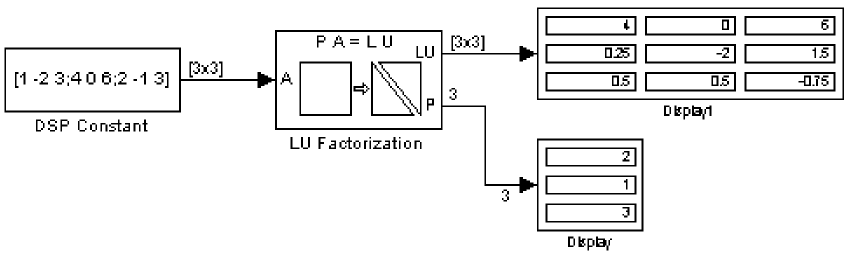
- QR Factorization
- Singular Value Decomposition

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]`.

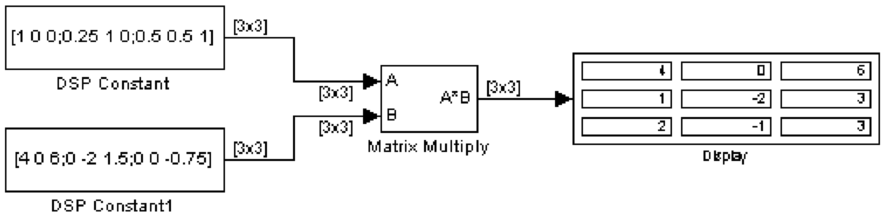
The lower output of the LU Factorization, `P`, is the permutation index vector, which indicates that the factored matrix A_p is generated from A by interchanging the first and second rows.

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

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:

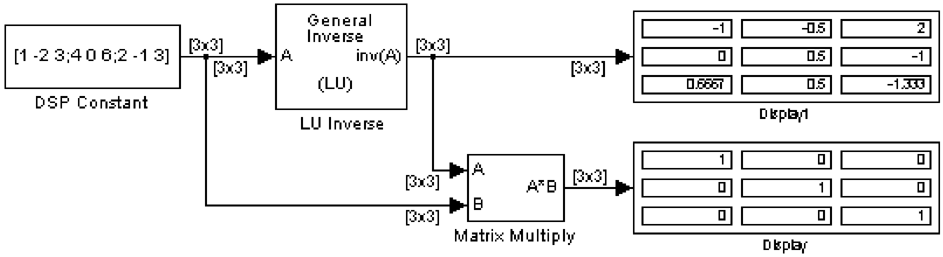
- Cholesky Inverse
- LDL Inverse
- LU Inverse
- Pseudoinverse

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

Block Reference

Blocks — By Category

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

Note To learn the basic concepts behind building DSP models with Simulink, see Chapter 2, “Simulink Review.” To find out about using blocks together for common DSP tasks, see Chapter 3, “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 “Blocks — Alphabetical List” on page 7-20.)

“DSP Sinks”	Various scopes and blocks for exporting signals to the MATLAB workspace
“DSP Sources”	Blocks that generate discrete-time signals such as sine waves and uniform random signals
“Estimation”	Linear prediction, parametric estimation, and power spectrum estimation blocks
“Filtering”	Digital filter design and implementation, adaptive, multirate, time-varying, and frequency-domain filters
“Math Functions”	Specialized math operations such as dB conversion and cumulative sum, matrix and linear algebra operations, and polynomial functions such as least squares polynomial fit
“Platform-Specific I/O”	Blocks for working with specific platforms such as sending audio data to standard audio devices on 32-bit Windows operating systems
“Quantizers”	A quantizer and uniform encoder and decoder
“Signal Management”	Buffers, blocks for selecting parts of a signal, blocks for modifying signal attributes such as frame status, and switches and counters

"Signal Operations"	Blocks such as Convolution, Downsample, Integer Delay, Unwrap, Zero Pad, and Window Function
"Statistics"	Correlation, Maximum, Mean, RMS, etc.
"Transforms"	Fast Fourier transform, discrete cosine transform, real and complex cepstrum, etc.

DSP Sinks

Various scopes and blocks for exporting signals to the MATLAB workspace. (Blocks are available in the DSP Sinks library.)

Display (Simulink block)	Show the value of the input
Matrix Viewer	Display a matrix as a color image
Signal To Workspace	Write simulation data to an array in the MATLAB workspace
Spectrum Scope	Compute and display the short-time FFT of each input signal
Time Scope (Simulink Block)	Display signals generated during a simulation
Triggered To Workspace	Write the input sample to an array in the MATLAB workspace when triggered
Vector Scope	Display a vector or matrix of time-domain, frequency-domain, or user-defined data

DSP Sources

Blocks that generate discrete-time signals such as sine waves and uniform random signals. (Blocks are available in the DSP Sources library.)

Chirp	Generate a swept-frequency cosine (chirp) signal
Constant Diagonal Matrix	Generate a square, diagonal matrix

Constant Ramp	Generate a ramp signal with length based on input dimensions
Discrete Impulse	Generate a discrete impulse
DSP Constant	Generate a discrete-time or continuous-time constant signal
Identity Matrix	Generate a 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 a specified number of sample times
Random Source	Generate randomly distributed values (Gaussian or uniform)
Signal From Workspace	Import a signal from the MATLAB workspace
Sine Wave	Generate a continuous or discrete sine wave
Triggered Signal From Workspace	Import signal samples from the MATLAB workspace when triggered

Estimation

The following sublibraries reside in the Estimation library:

- "Linear Prediction"
- "Parametric Estimation"
- "Power Spectrum Estimation"

Linear Prediction

Blocks for linear prediction and working with linear prediction coefficients. (Blocks are available in the Linear Prediction library.)

Autocorrelation LPC	Determine the coefficients of an Nth-order forward linear predictor
LPC to LSF/LSP Conversion	Convert linear prediction coefficients (LPCs) to line spectral pairs (LSPs) or line spectral frequencies (LSFs)
LSF/LSP to LPC Conversion	Convert line spectral pairs (LSPs) or line spectral frequencies (LSFs) to linear prediction coefficients (LPCs)

Parametric Estimation

Blocks for computing estimates of autoregressive model parameters using various methods. (Blocks are available in the Parametric Estimation library.)

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

Power Spectrum Estimation

Blocks for computing parametric and nonparametric spectral estimates using various methods. (Blocks are available in the Power Spectrum Estimation library.)

Burg Method	Compute a parametric spectral estimate using the Burg method
Covariance Method	Compute a parametric spectral estimate using the covariance method
Magnitude FFT	Compute a nonparametric estimate of the spectrum using the periodogram method
Modified Covariance Method	Compute a parametric spectral estimate using the modified covariance method
Short-Time FFT	Compute a nonparametric estimate of the spectrum using the short-time, fast Fourier transform (ST-FFT) method
Yule-Walker Method	Compute a parametric estimate of the spectrum using the Yule-Walker AR method

Filtering

The following sublibraries reside in the Filtering library:

- “Adaptive Filters”
- “Filter Design, Analysis, and Implementation”
- “Multirate Filters”

Adaptive Filters

Blocks for computing filter estimates of an input using various algorithms. (Blocks are available in the Adaptive Filters library.)

Kalman Adaptive Filter	Compute filter estimates for an input using the Kalman adaptive filter algorithm
LMS Adaptive Filter	Compute filter estimates for an input using the LMS adaptive filter algorithm
RLS Adaptive Filter	Compute filter estimates for an input using the RLS adaptive filter algorithm

Filter Design, Analysis, and Implementation

Blocks for designing, analyzing, and implementing various filters. (Blocks are available in the Filter Designs library.)

Analog Filter Design	Design and implement an analog filter
Digital Filter	Filter inputs with a specified time-varying or static digital FIR or IIR filter
Digital Filter Design	Design, analyze, and implement a variety of digital FIR and IIR filters.
Filter Realization Wizard	Automatically construct filter realizations using Sum, Gain, and Unit Delay blocks
Overlap-Add FFT Filter	Implement the overlap-add method of frequency-domain filtering
Overlap-Save FFT Filter	Implement the overlap-save method of frequency-domain filtering

Multirate Filters

Blocks for implementing various multirate filters. (Blocks are available in the Multirate Filters library.)

Dyadic Analysis Filter Bank	Decompose a signal into components of equal or logarithmically decreasing frequency subbands and sample rates
Dyadic Synthesis Filter Bank	Reconstruct a signal from its multirate bandlimited components.
FIR Decimation	Filter and downsample an input signal
FIR Interpolation	Upsample and filter an input signal
FIR Rate Conversion	Upsample, filter, and downsample an input signal
Two-Channel Analysis Subband Filter	Decompose a signal into a high-frequency subband and a low-frequency subband
Two-Channel Synthesis Subband Filter	Reconstruct a signal from a high-frequency subband and a low-frequency subband

Math Functions

The following sublibraries reside in the Math Functions library:

- “Math Operations”
- “Matrices and Linear Algebra”
- “Polynomial Functions”

Math Operations

Blocks for specialized math operations not provided in the Simulink math library. (Blocks are available in the Math Operations library.)

Complex Exponential	Compute the complex exponential function
Cumulative Product	Compute the cumulative product of row or column elements
Cumulative Sum	Compute the cumulative sum of row or column elements
dB Conversion	Convert magnitude data to decibels (dB or dBm)
dB Gain	Apply a gain specified in decibels
Difference	Compute the element-to-element difference along rows or columns
Normalization	Normalize an 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"
- "Matrix Factorizations"
- "Matrix Inverses"
- "Matrix Operations"

Linear System Solvers. Blocks that solve the matrix equation AX = B for X using various methods. (Blocks are available in the Linear System Solvers library.)

Backward Substitution	Solve the equation U*X*=B for *X* when U is an upper triangular matrix.
Cholesky Solver	Solve the equation SX = B for X when S is a square Hermitian positive definite matrix

Forward Substitution	Solve the equation LX = B for X when L is a lower triangular matrix
LDL Solver	Solve the equation SX = B for X when S is a square Hermitian positive definite matrix
Levinson-Durbin	Solve a linear system of equations using Levinson-Durbin recursion
LU Solver	Solve the equation AX = B for X when A is a square matrix
QR Solver	Find a minimum-norm-residual solution to the equation AX=B
SVD Solver	Solve the equation AX=B using singular value decomposition

Matrix Factorizations. Blocks for factoring matrices using various methods. (Blocks are available in the Matrix Factorizations library.)

Cholesky Factorization	Factor a square Hermitian positive definite matrix into triangular components
LDL Factorization	Factor a square Hermitian positive definite matrix into lower, upper, and diagonal components
LU Factorization	Factor a square matrix into lower and upper triangular components
QR Factorization	Factor a rectangular matrix into unitary and upper triangular components
Singular Value Decomposition	Factor a matrix using singular value decomposition

Matrix Inverses. Blocks for inverting matrices using various methods. (Blocks are available in the Matrix Inverses library.)

Cholesky Inverse	Compute the inverse of a Hermitian positive definite matrix using Cholesky factorization
LDL Inverse	Compute the inverse of a Hermitian positive definite matrix using LDL factorization
LU Inverse	Compute the inverse of a square matrix using LU factorization
Pseudoinverse	Compute the Moore-Penrose pseudoinverse of a matrix

Matrix Operations. Blocks for various matrix operations such as extracting the diagonal, overwriting matrix values, and multiplying matrices. (Blocks are available in the Matrix Operations library.)

Constant Diagonal Matrix	Generate a square, diagonal matrix
Create Diagonal Matrix	Create a square diagonal matrix from diagonal elements
Extract Diagonal	Extract the main diagonal of the input matrix
Extract Triangular Matrix	Extract the lower or upper triangle from an input matrix
Identity Matrix	Generate a matrix with ones on the main diagonal and zeros elsewhere
Matrix Concatenation (Simulink block)	Concatenate inputs horizontally or vertically
Matrix 1-Norm	Compute the 1-norm of a matrix
Matrix Multiply	Multiply input matrices
Matrix Product	Multiply the elements of a matrix along rows or columns

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

Polynomial Functions

Blocks for working with polynomials. (Blocks are available in the Polynomial Functions library.)

Least Squares Polynomial Fit	Compute the coefficients of the polynomial that best fits the input data in a least-squares sense
Polynomial Evaluation	Evaluate a polynomial expression
Polynomial Stability Test	Determine whether all roots of the input polynomial are inside the unit circle using the Schur-Cohn algorithm

Platform-Specific I/O

Windows (WIN32)

Blocks for working with audio data in 32-bit Windows operating systems. (Blocks are available in the Windows (WIN32) library.)

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

Quantizers

Blocks for quantizing data. (Blocks are available in the Quantizers library.)

Quantizer (Simulink block)	Discretize input at a specified interval
Uniform Decoder	Decode an integer input to a floating-point output
Uniform Encoder	Quantize and encode a floating-point input to an integer output

Signal Management

The following sublibraries reside in the Signal Management library:

- "Buffers"
- "Indexing"
- "Signal Attributes"

- "Switches and Counters"

Buffers

Blocks for changing the sample rate or frame rate of a signal by accumulating input samples before outputting them. (Blocks are available in the Buffers library.)

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

Indexing

Blocks for manipulating the ordering of a signal such as selecting parts of a signal or flipping a signal. (Blocks are available in the Indexing library.)

Flip	Flip the input vertically or horizontally
Multiport Selector	Distribute arbitrary subsets of input rows or columns to multiple output ports
Selector (Simulink block)	Select input elements from a vector or matrix signal
Submatrix	Select a subset of elements (submatrix) from a matrix input
Variable Selector	Select a subset of rows or columns from the input

Signal Attributes

Blocks for inspecting or modifying signal attributes such as frame status and complexity. (Blocks are available in the Signal Attributes library.)

Check Signal Attributes	Generate an error when the input signal does or does not match selected attributes exactly
Convert 1-D to 2-D	Reshape a 1-D or 2-D input to a 2-D matrix with the specified dimensions
Convert 2-D to 1-D	Convert a 2-D matrix input to a 1-D vector
Data Type Conversion (Simulink block)	Convert input signal to specified data type
Frame Status Conversion	Specify the frame status of the output, sample-based or frame-based
Inherit Complexity	Change the complexity of the input to match that of a reference signal

Switches and Counters

Blocks for performing an action when an event such as a threshold crossing in the data occurs. (Blocks are available in the Switches and Counters library.)

Counter	Count up or down through a specified range of numbers
Edge Detector	Detect a transition of the input 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 a specified number of sample times
N-Sample Switch	Switch between two inputs after a specified number of sample periods

Signal Operations

Blocks for performing the following types of operations on a signal. (Blocks are available in the Signal Operations library.):

- "Delaying"
- "Padding"
- "Resampling"
- "Miscellaneous Operations"

Delaying

Integer Delay	Delay an input by an integer number of sample periods
Variable Fractional Delay	Delay an input by a time-varying fractional number of sample periods
Variable Integer Delay	Delay the input by a time-varying integer number of sample periods

Padding

Pad	Alter the input size by padding or truncating rows and/or columns
Zero Pad	Alter the input size by zero-padding or truncating rows and/or columns

Resampling

Downsample	Resample an input at a lower rate by deleting samples
Repeat	Resample an input at a higher rate by repeating values
Upsample	Resample an input at a higher rate by inserting zeros

Miscellaneous Operations

Convolution	Compute the convolution of two inputs
Interpolation	Interpolate values of real input samples
Unwrap	Unwrap the phase of a signal
Window Function	Compute a window, and/or apply a window to an input signal
Sample and Hold	Sample and hold an input signal

Statistics

Blocks for performing various statistical computations. (Blocks are available in the Statistics library.)

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

Transforms

Blocks for computing various transforms. (Blocks are available in the Transforms library.)

Analytic Signal	Compute the analytic signal of a discrete-time input
Complex Cepstrum	Compute the complex cepstrum of an input
DCT	Compute the discrete cosine transform (DCT) of the input
DWT	Compute the discrete wavelet transform (DWT) of the input signal
FFT	Compute the fast Fourier transform (FFT) of the input
IDCT	Compute the inverse discrete cosine transform (IDCT) of the input
IDWT	Compute the inverse discrete wavelet transform (IDWT) of the input signal
IFFT	Compute the inverse fast Fourier transform (IFFT) of the input
Magnitude FFT	Compute a nonparametric estimate of the spectrum using the periodogram method.
Real Cepstrum	Compute the real cepstrum of an input

Blocks — Alphabetical List

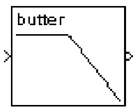
Purpose

Design and implement an analog filter.

Library

Filtering / Filter Designs

Description



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.

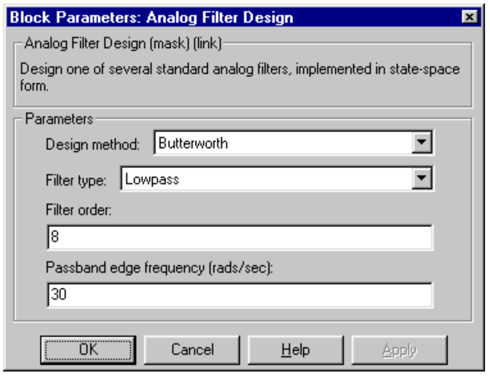
Filter Design	Description
Butterworth	The magnitude response of a Butterworth filter is maximally flat in the passband and monotonic overall.
Chebyshev type I	The magnitude response of a Chebyshev type I filter is equiripple in the passband and monotonic in the stopband.
Chebyshev type II	The magnitude 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 table below 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 a portion 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 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 to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

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

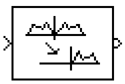
See the following sections for related information:

- "Filters" on page 4-1
- "Analog IIR Filters" on page 4-35
- "Filtering" on page 7-6 — List of all DSP Blockset filtering blocks

Purpose Compute the analytic signal of a discrete-time input.

Library Transforms

Description The Analytic Signal block computes the complex analytic signal corresponding to each channel of the real M-by-N input, u.



$$y = u + jH\{u\}$$

where $j = \sqrt{-1}$ and $H\{\cdot\}$ 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.

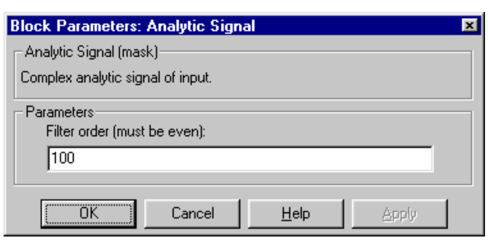
Sample-Based Operation

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 to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Also see “Transforms” on page 7-19 for a list of all the blocks in the Transforms library.

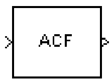
Purpose

Compute the autocorrelation of a vector input.

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 $y_{1,j}$ is the zero-lag element in the jth column. When **Compute all non-negative lags** is selected, l=M. Otherwise, l is specified as a nonnegative integer by the **Maximum positive lag** 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.

If 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 **Scaling** 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.

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

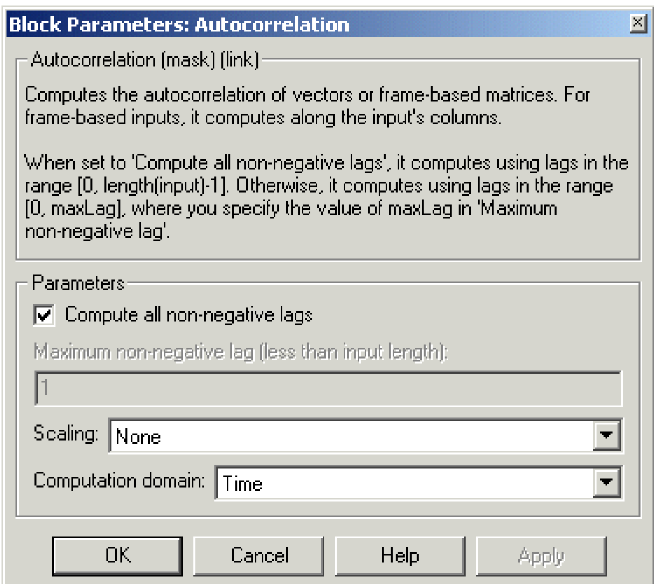
- **Unity at zero-lag** — Normalizes the estimate of the autocorrelation for each channel so that the zero-lag sum is identically 1.

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

The **Computation domain** parameter sets the domain in which the block computes convolutions to one of the following settings:

- **Time** — The block computes in the time domain, which minimizes memory use.
- **Frequency** — The block computes in the frequency domain, which may require fewer computations than computing in the time domain (depending on the input length).

Dialog Box



Compute all non-negative lags

When selected, computes the autocorrelation over all non-negative lags in the range [0, `length(input)-1`].

Maximum positive lag

The maximum positive lag, l, for the autocorrelation. This parameter is enabled when the **Compute all non-negative lags** check box is unselected.

Scaling

The type of scaling for the autocorrelation: **None**, **Biased**, **Unbiased**, or **Unity at zero-lag**. Tunable, except in the Simulink external mode.

Computation domain

The domain in which the block computes convolutions: **Time** or **Frequency**. To minimize memory use, select **Time**; frequency domain computations may require fewer computations than those in the time domain depending on the input length.

Supported Data Types

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

To learn how to convert to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

Correlation	DSP Blockset
`xcorr`	Signal Processing Toolbox

Also see "Statistics" on page 7-18 for a list of all the blocks in the Statistics library.

Autocorrelation LPC

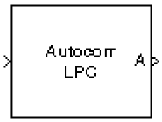
Purpose

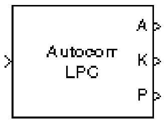
Determine the coefficients of an Nth-order forward linear predictor.

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 **Inherit prediction order from input dimensions** is selected, the prediction order, N, is inherited from the input dimensions. Otherwise, the **Prediction order** parameter sets the value of N.

When **Output(s)** is set to **A**, port `A` is enabled. Port `A` outputs an (N+1)-by-1 column vector, $a = [1\ a_2\ a_3\ \ldots\ a_{N+1}]^T$, 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.

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

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 **Output prediction error power (P)** is selected, port `P` is enabled. The prediction error power, a scalar, is output at port `P`.

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

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

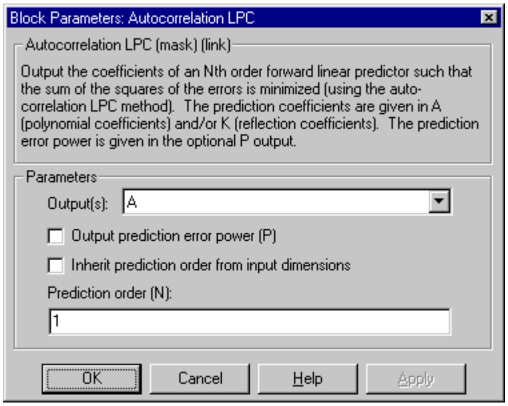
leads to the system of equations

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

where $r = [r_1\ r_2\ r_3\ \ldots\ r_{n+1}]^T$ 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 **Inherit prediction order from input dimensions** is selected.

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 to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

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

Also see "Linear Prediction" on page 7-5 for a list of all the blocks in the Linear Prediction library.

Backward Substitution

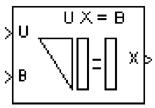
Purpose

Solve the equation U*X*=B for *X* when U is an upper triangular matrix.

Library

Math Functions / Matrices and Linear Algebra / Linear System Solvers

Description

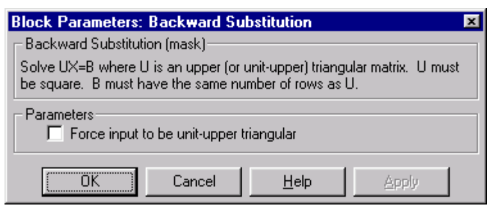


The Backward Substitution block solves the linear system U*X*=B by simple backward substitution of variables, where U is the upper triangular M-by-M matrix input to the `U` port, and B is the M-by-N matrix input to the `B` port. The output is the solution of the equations, the M-by-N matrix *X*, and 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 **Force input to be unit-upper triangular** is selected, the block replaces the elements on the diagonal of U with ones. 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



Force input to be unit-upper triangular

Replaces the elements on the diagonal of U with 1s when selected. Tunable in simulation.

Supported Data Types

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

To learn how to convert to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Cholesky Solver	DSP Blockset
Forward Substitution	DSP Blockset
LDL Solver	DSP Blockset
Levinson-Durbin	DSP Blockset
LU Solver	DSP Blockset
QR Solver	DSP Blockset

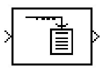
See "Solving Linear Systems" on page 6-7 for related information. Also see "Linear System Solvers" on page 7-9 for a list of all the blocks in the Linear System Solvers library.

Buffer

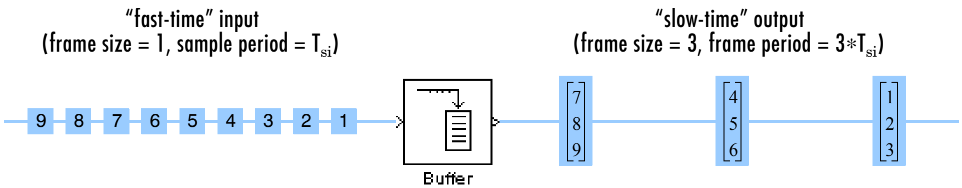
Purpose Buffer the input sequence to a smaller or larger frame size.

Library Signal Management / Buffers

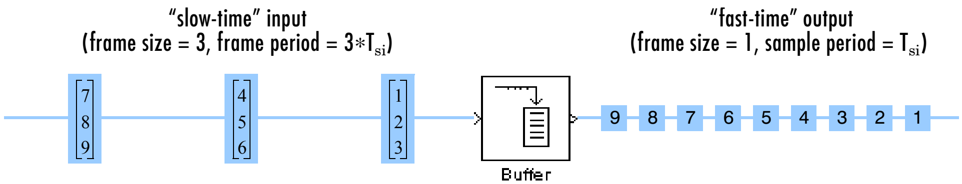
Description



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.



Buffering to a smaller frame size yields an output with a *faster* frame rate than the input, as illustrated below for scalar output.



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, $T_{so} = T_{si}$.

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.

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

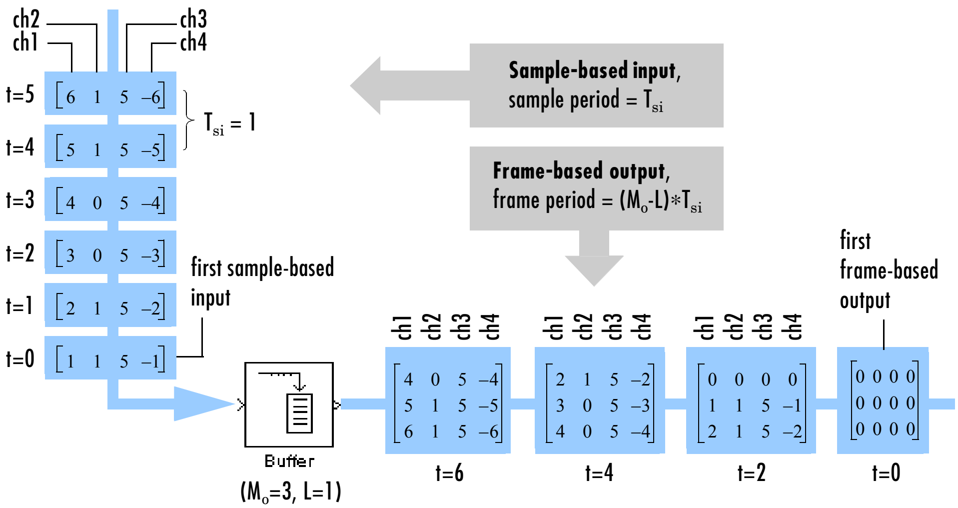
M_o=1, the input is simply passed through to the output, and retains the same dimension.

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 $(M_o\text{-}L)*T_{si}$, 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 $(M_o\text{-}L)*T_{si}$, 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.



Note 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 (see "Latency" below). 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 above example,

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

d =
      4
```

This agrees with the four samples of delay (zeros) per channel shown in the figure above.

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 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 (i.e., 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.

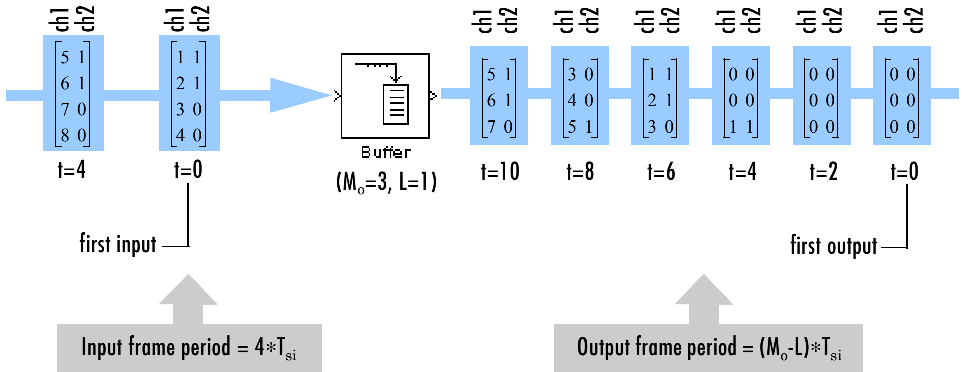
The **Buffer overlap** parameter, L, specifies the number of samples (rows) from the current output to repeat in the next output, where $L < M_o$. For $0 \leq L < M_o$, the number of *new* input samples the block acquires before propagating the buffered data to the output is the difference between the **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 $(M_o\text{-}L) * T_{si}$, 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.



Note 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 (see “Latency” below). 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.

Latency

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. 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

$$D = \begin{cases} M_o + L & (L \geq 0) \\ M_o & (L < 0) \end{cases}$$

If 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:

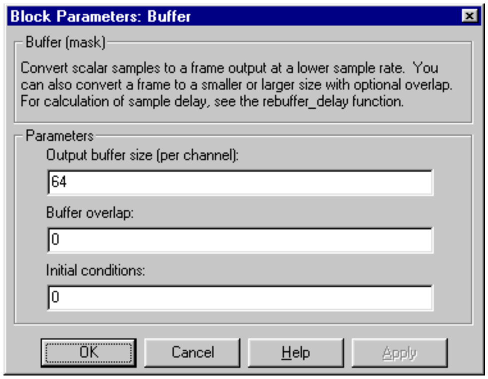
- An M_o-by-N matrix
- A length-M_o vector to be repeated across all columns of the initial output(s)
- A scalar to be repeated across all elements of the initial output(s)

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)

See "Excess Algorithmic Delay (Tasking Latency)" on page 3-91 and "The Simulation Parameters Dialog Box" in the Simulink documentation for more information about block rates and the Simulink tasking modes.

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.

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 to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

Delay Line	DSP Blockset
Unbuffer	DSP Blockset
`rebuffer_delay`	DSP Blockset

See “Buffering Sample-Based and Frame-Based Signals” on page 3-47 for related information. Also see “Buffers” on page 7-14 for a list of all the blocks in the Buffers library.

Purpose

Compute an estimate of AR model parameters using the Burg method.

Library

Estimation / Parametric Estimation

Description

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 **Inherit estimation order from input dimensions** is selected, 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,

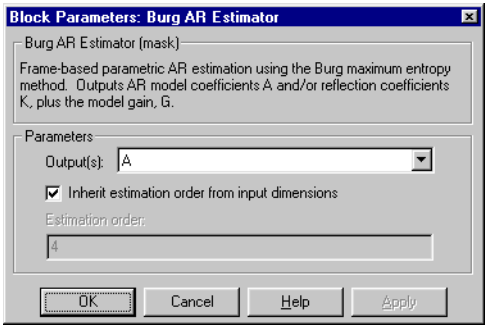
  ```
  [1 a(2) ... a(p+1)]
  ```

- **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).
- **A and K** — The block outputs both realizations.

The scalar gain, G, is provided at the bottom output (`G`).

Burg AR Estimator

Dialog Box



Output(s)

The realization to output, model coefficients, reflection coefficients, or both.

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 **Inherit estimation order from input dimensions** is not selected.

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

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

To learn how to convert to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Burg Method	DSP Blockset
Covariance AR Estimator	DSP Blockset
Modified Covariance AR Estimator	DSP Blockset
Yule-Walker AR Estimator	DSP Blockset
`arburg`	Signal Processing Toolbox

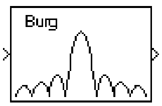
Also see "Parametric Estimation" on page 7-5 for a list of all the blocks in the Parametric Estimation library.

Burg Method

Purpose Compute a parametric spectral estimate using the 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 **Inherit estimation order from input dimensions** is selected, 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 **Inherit FFT length from estimation order** is selected, N_{fft} is specified by the frame size of the input, which must be a power of 2. When **Inherit FFT length from estimation order** is *not* selected, 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.

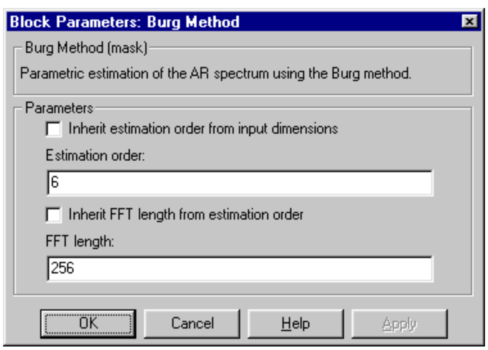
	Burg	**Covariance**	**Modified Covariance**	**Yule-Walker**
Characteristics	Does not apply window to data	Does not apply window to data	Does not apply window to data	Applies window to data
	Minimizes the forward and backward prediction errors in the least-squares sense, with the AR coefficients constrained 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	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. Tunable.

Estimation order

The order of the AR model. This parameter is enabled when **Inherit estimation order from input dimensions** is not selected.

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. Tunable.

FFT length

The number of data points, N_{fft}, on which to perform the FFT. If N_{fft} exceeds the input frame size, the frame is zero-padded as needed. This parameter is enabled when **Inherit FFT length from input dimensions** is not selected.

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.

Supported Data Types

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

To learn how to convert to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Burg AR Estimator	DSP Blockset
Covariance Method	DSP Blockset
Modified Covariance Method	DSP Blockset
Short-Time FFT	DSP Blockset
Yule-Walker Method	DSP Blockset
`pburg`	Signal Processing Toolbox

See “Power Spectrum Estimation” on page 6-6 for related information. Also see a list of all blocks in the Power Spectrum Estimation library.

Check Signal Attributes

Purpose

Generate an error when the 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 if 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 if 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 **Ignore** is selected 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 also be displayed in a model by attaching a Probe block with **Probe complex signal** selected, or by selecting **Port data types** from the model window's **Format** menu.)
- **Frame status**

  Checks whether the signal is frame-based or sample-based. (Note that Simulink displays sample-based signals using a single line, $\rightarrow$, and frame-based signals using a double line, $\Rightarrow$.)

- **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. See“Signal Dimension Nomenclature” on page 1-13 for a description of Simulink signal dimensions. 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
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

Dimensions (Continued)	Is...	Is not...
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, 1-by-1 matrix (2-D scalar)	M-by-N matrix with M≠N, 1-D vector, 1-by-N matrix (row vector), M-by-1 matrix (column vector)

Note that when **Signal dimensions** is selected from the model window **Format** menu, 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
Fixed-point	fixed-point	`boolean`, `uint8`, `int8`, `uint16`, `int16`, `uint32`, `int32`, `single`, `double`

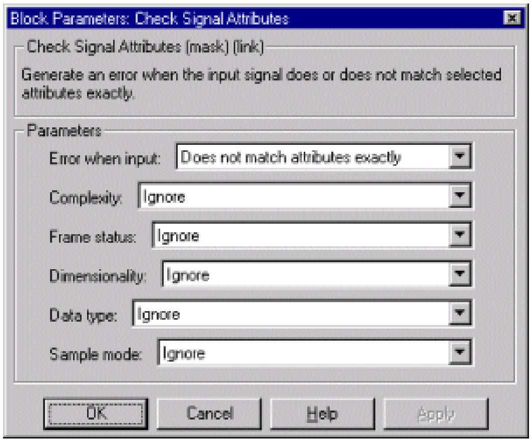
Integer	**Signed integer**
	`int8`, `int16`, `int32`
	Unsigned integer
	`uint8`, `uint16`, `uint32`
		`boolean`, `single`, `double`

Note that data type information can also be displayed in a model by selecting **Port data types** from the model window's **Format** menu.

- **Sample mode**

  Checks whether the signal is discrete-time or continuous-time. (Note that when **Sample time colors** is selected from the **Format** menu, 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.)

Dialog Box



Error if 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**.

Frame status

The frame status for which the input should be checked, **Sample-based** or **Frame-based**.

Dimensionality

Specifies whether the input should be checked for compliance (**Is...**) or noncompliance (**Is not...**) with the attributes in the subordinate **Dimension** menu.

Dimensions

The dimensions for which the input should be checked. This parameter is available when **Is...** or **Is not...** is selected in the **Dimensionality** menu.

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.

General data type

The general data type for which the input should be checked. This parameter is available when **Is...** or **Is not...** is selected in the **Data type** menu, and enables the subordinate **Specific data type** parameter in most cases.

Specific data type

The specific data type for which the input should be checked. This parameter is available when **Floating-point**, **Fixed-point**, or **Integer** is selected in the **General data type** menu.

Sample mode

The sample mode for which the input should be checked, **Discrete** or **Continuous**.

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 to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

Buffer	DSP Blockset
Convert 1-D to 2-D	DSP Blockset
Convert 2-D to 1-D	DSP Blockset
Data Type Conversion	Simulink
Frame Status Conversion	DSP Blockset
Inherit Complexity	DSP Blockset
Probe	Simulink
Reshape	Simulink
Submatrix	DSP Blockset

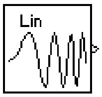
Also see “Signal Attributes” on page 7-15 for a list of all the blocks in the Signal Attributes library.

Chirp

Purpose Generate a swept-frequency cosine (chirp) signal.

Library DSP 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 table. The **Sample time** parameter sets the sample time of both sample- and frame-based outputs.

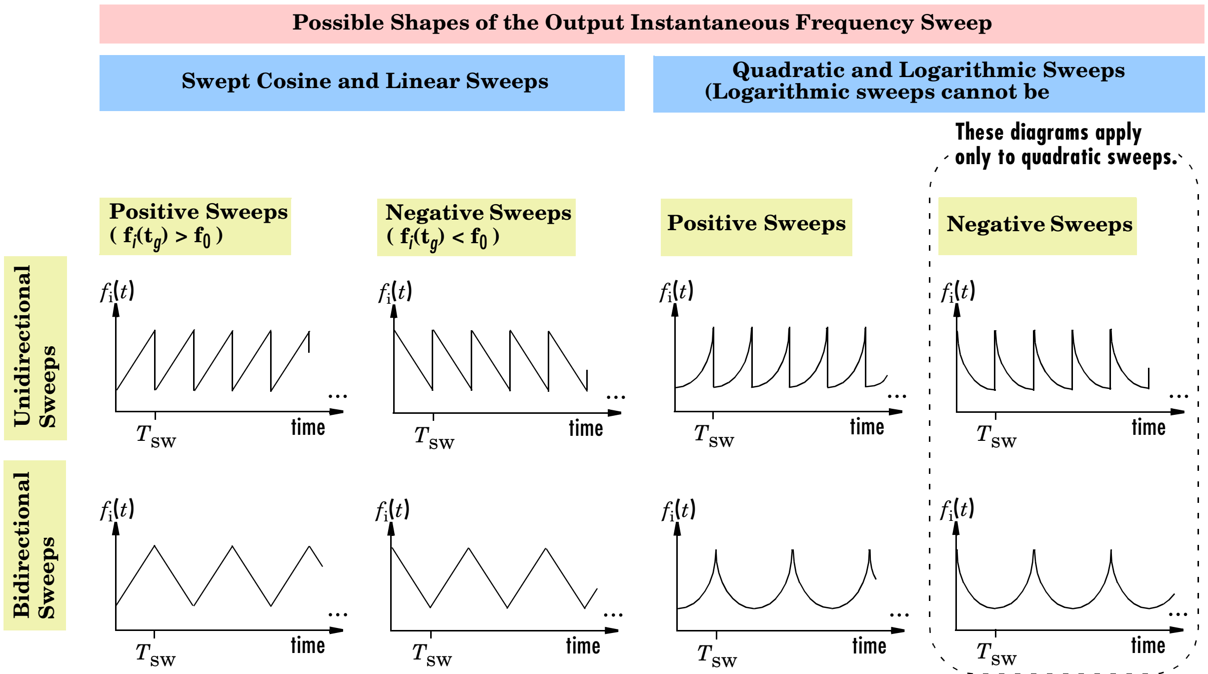
Setting of Samples Per Frame Parameter	Output Frame Status
1	Sample-based
n (any integer greater than 1)	Frame-based, frame size n

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".

The following diagram illustrates the possible shapes of the frequency sweep that you can obtain by setting the **Frequency sweep** and **Sweep mode** parameters.



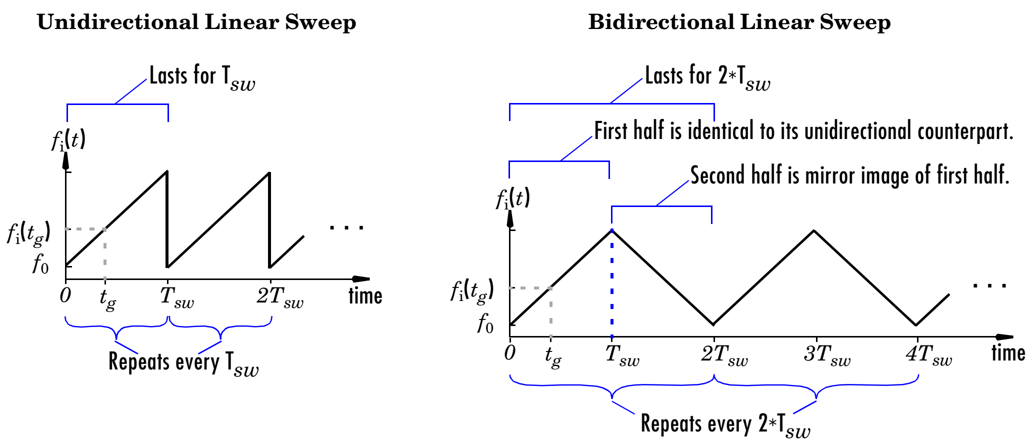
For information on how to set the frequency values in your sweep, see “Setting Instantaneous Frequency Sweep Values”.

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 7-64). 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**, $2*T_{sw}$ • Repeats once every $2*T_{sw}$ • 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".



Setting Instantaneous Frequency Sweep Values

Set the following parameters to tune the frequency values of your output frequency sweep:

- **Initial frequency** (Hertz), f_0

- **Target frequency** (Hertz), $f_i(t_g)$
- **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”.

Table 7-1: 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	$f_0 + 1$	$f_i(t_g)$	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”
- “Output Computation Method for Linear, Quadratic, and Logarithmic Frequency Sweeps”
- “Output Computation Method for Swept Cosine Frequency Sweep”

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 $f_i(0) < f_i(t_g)$. To derive equations for other cases, you may 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 7-64.

Table 7-2 contains the following variables:

- $f_i(t)$ — the user-specified frequency sweep
- $f_{i(actual)}(t)$ — the actual output frequency sweep, usually equal to $f_i(t)$
- $y_{chirp}(t)$ — 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 $y_{chirp}(0) = \cos(\phi_0)$

Table 7-2: 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	$y_{chirp}(t) = \cos(\psi(t) + \phi_0)$	$f_i(t) = f_0 + \beta t$	$\beta = \frac{f_i(t_g) - f_0}{t_g}$	$f_{i(actual)}(t) = f_i(t)$
Quadratic	Same as Linear	$f_i(t) = f_0 + \beta t^2$	$\beta = \frac{f_i(t_g) - f_0}{t_g^2}$	$f_{i(actual)}(t) = f_i(t)$
Logarithmic	Same as Linear	$f_i(t) = f_0 + 10^{\beta t}$ Note $f_i(0) = f_0 + 1$	$\beta = \frac{\log[f_i(t_g) - f_0]}{t_g}$ Where $f_i(t_g) > f_0$	$f_{i(actual)}(t) = f_i(t)$
Swept cosine	$y_{chirp}(t) = \cos(2\pi f_i(t)t + \phi_0)$	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 $\psi(t)$

$$f_i(t) = \frac{1}{2\pi} \cdot \frac{d\psi(t)}{dt} \tag{7-1}$$

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. 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 7-67.

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 (Equation 7-1 does not hold.)

Note that Equation 7-1 does not hold for the swept cosine chirp, so the user-defined frequency sweep, $f_i(t)$, is not the actual output frequency sweep, $f_{i(actual)}(t)$, of the swept cosine chirp. Thus, the swept cosine output may not behave as you expect. To learn more about swept cosine chirp behavior, see "Cautions Regarding the Swept Cosine Sweep" on page 7-69 and "Equations for Output Computation" on page 7-67.

Cautions Regarding the Swept Cosine Sweep

If 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 may have unexpected frequency content. For details, see the following two sections.

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 7-69. See Table 7-1, Instantaneous Frequency Sweep Values, 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 may be closer to $1/T_{sw}$, far exceeding the **Initial frequency** and **Target frequency** parameter values.

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"
- "Example 2: Bidirectional Sweeps"
- "Example 3: When Sweep Time is Greater Than Target Time"

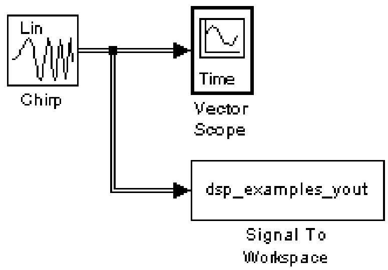
Examples 4 and 5 illustrate Chirp block settings that may produce unexpected outputs:

- "Example 4: Output Sweep with Negative Frequencies"
- "Example 5: Output Sweep with Frequencies Greater Than Half the Sampling Frequency"

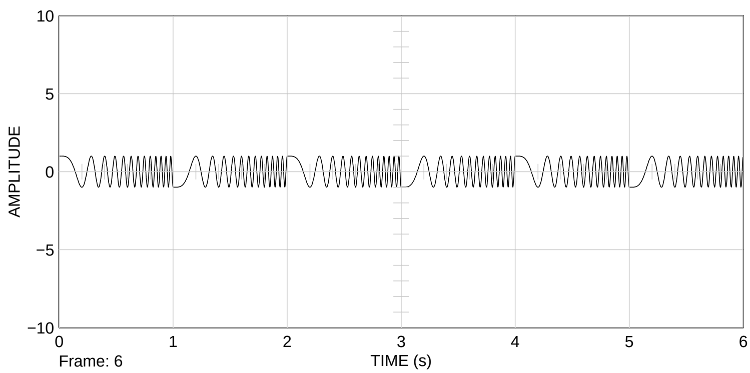
Example 1: Setting a Final Frequency Value for Unidirectional Sweeps. Often times, you may want a unidirectional sweep for which you know the initial and final frequency values. You can specify the final frequency of a unidirectional sweep by setting **Target time** equal to **Sweep time**, in which case the **Target frequency** becomes the final frequency in the sweep. The following model demonstrates this method.

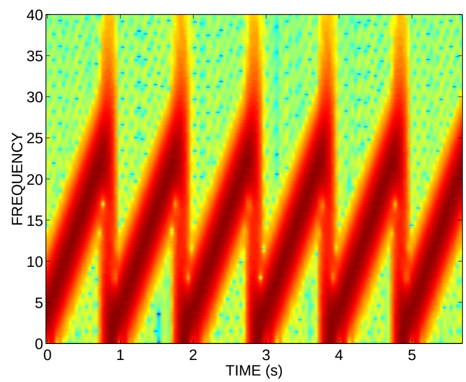
This technique may not work for swept cosine sweeps. For details, see "Cautions Regarding the Swept Cosine Sweep" on page 7-69.

Open the Example 1 model by clicking here in the MATLAB Help Browser. 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 Hertz`) 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.

```
specgram(dsp_examples_yout,[0:.01:40],400,hamming(128),110)
```

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

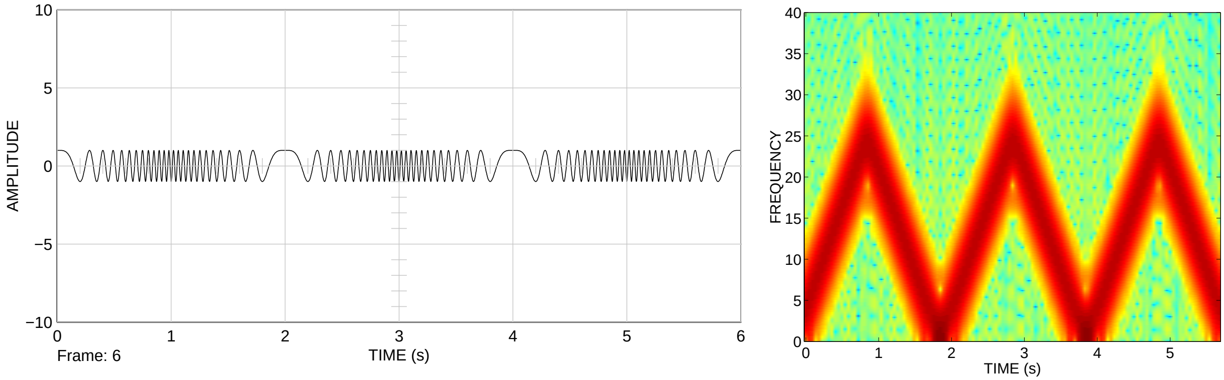
Variable name	dsp_examples_yout

Simulation Parameters 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 clicking here in the MATLAB Help Browser.



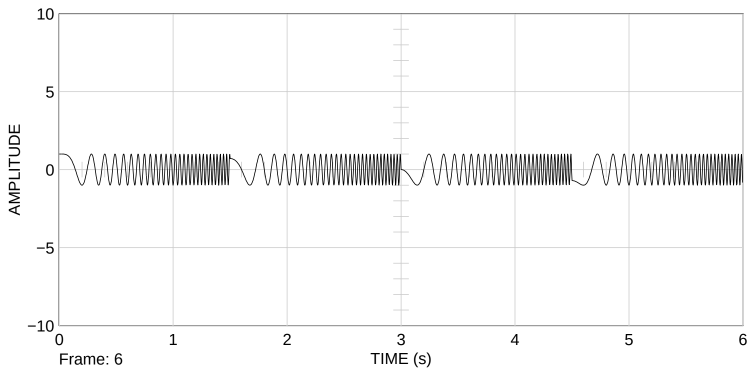
Run your model to see the time domain output, and then type the following command to view the chirp output spectrogram.

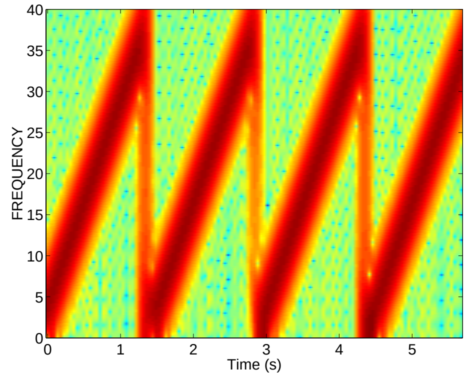
```
specgram(dsp_examples_yout,[0:.01:40],400,hamming(128),110)
```

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 Hertz) 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 may arise when you set **Sweep time** greater than **Target time**; see "Example 4: Output Sweep with Negative Frequencies" for details.

Open the Example 3 model by clicking here in the MATLAB Help Browser.





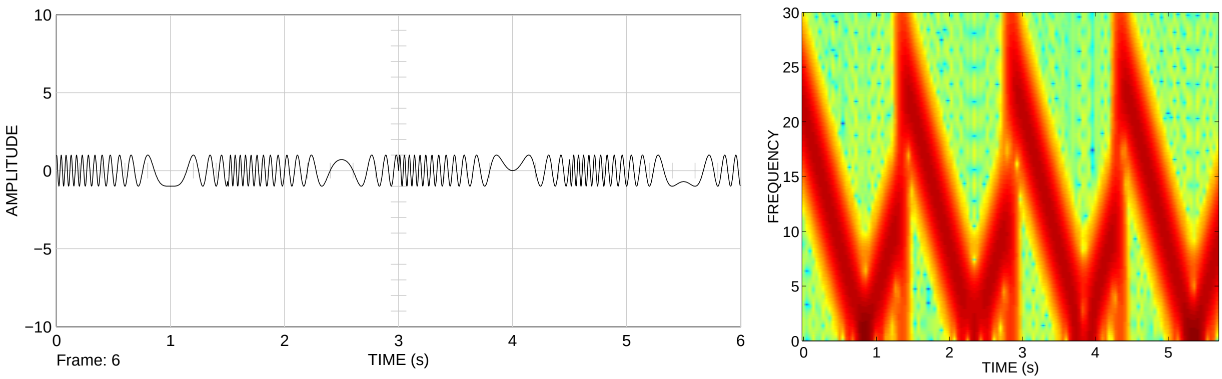
Run your model to see the time domain output, and then type the following command to view the chirp output spectrogram.

```
specgram(dsp_examples_yout,[0:.01:40],400,hamming(128),110)
```

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 may 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 Hertz 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 may reflect negative sweep frequencies along the *x*-axis so they appear to be positive, as in the one below. If you unexpectedly get a chirp output with a spectrogram resembling the one following, your chirp's sweep may contain negative frequencies. See the next example for another possible unexpected chirp output.

Open the Example 4 model by clicking here in the MATLAB Help Browser.

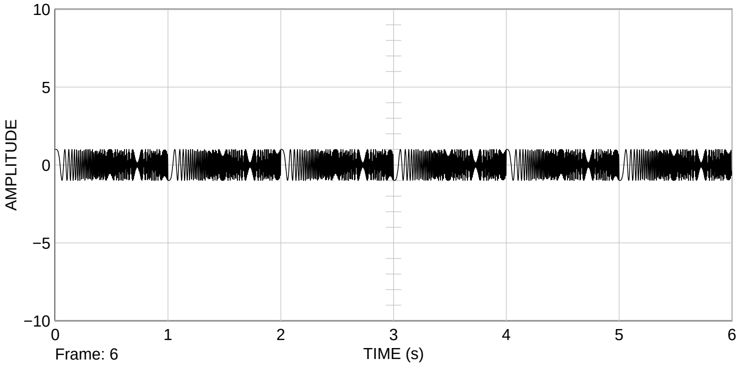


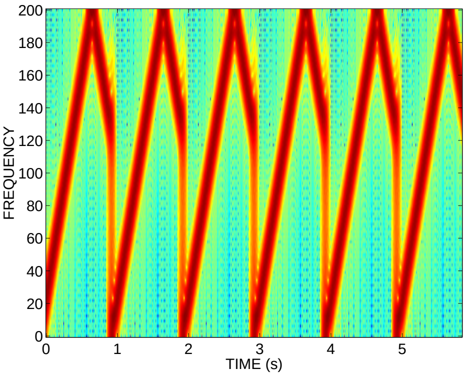
Run your model to see the time domain output, and then type the following command to view the chirp output spectrogram.

```
specgram(dsp_examples_yout,[0:.1:30],400,hamming(128),110);
```

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 may not behave as you expect* because the sweep contains frequencies greater than half the sampling frequency (`200 Hertz`), which causes aliasing. If you unexpectedly get a chirp output with a spectrogram resembling the one following, your chirp's sweep may contain frequencies greater than half the sampling frequency. See the previous example for another possible unexpected chirp output.

Open the Example 5 model by clicking here in the MATLAB Help Browser.

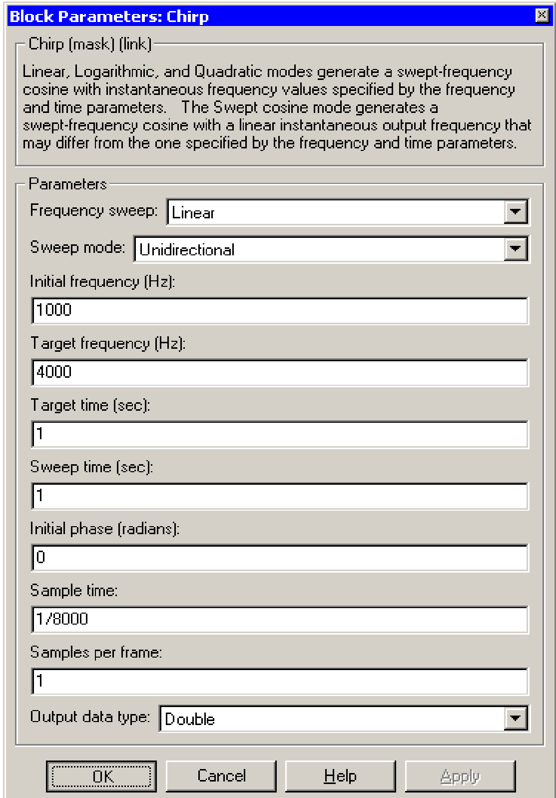




Run your model to see the time domain output, and then type the following command to view the chirp output spectrogram.

```
specgram(dsp_examples_yout,256,400,hamming(64),60)
```

Dialog Box



Frequency sweep

The type of output instantaneous frequency sweep, $f_i(t)$: **Linear**, **Logarithmic**, **Quadratic, or Swept cosine**. Tunable.

Sweep mode

The directionality of the chirp signal: **Unidirectional** or **Bidirectional**. Tunable.

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**, T_{sw}, 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.

Supported Data Types

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

To learn how to convert to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Signal From Workspace	DSP Blockset
Signal Generator	Simulink
Sine Wave	DSP Blockset
`chirp`	Signal Processing Toolbox
`specgram`	Signal Processing Toolbox

Also see the following topics:

- "Creating Signals Using Signal Generator Blocks" on page 3-36 — How to use this and other blocks to generate signals
- "DSP Sources" on page 7-3 — List of all blocks in the DSP Sources library

Cholesky Factorization

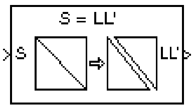
Purpose

Factor a square Hermitian positive definite matrix into triangular components.

Library

Math Functions / Matrices and Linear Algebra / Matrix Factorizations

Description

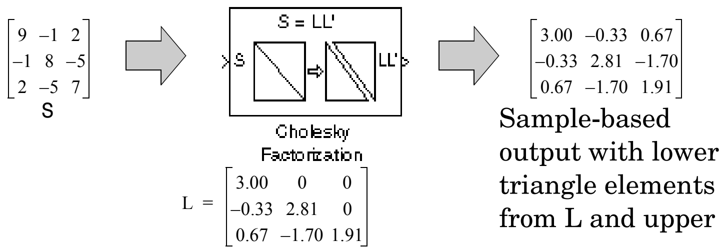


The Cholesky Factorization block uniquely factors the square Hermitian positive definite input matrix S as

$$S = LL^*$$

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.

Block Output Composed of L and L*



Input Requirements for Valid Output

The block output is valid only if its input has the following characteristics:

Note • 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 Non-Positive Definite Input".

Response to Non-Positive Definite Input

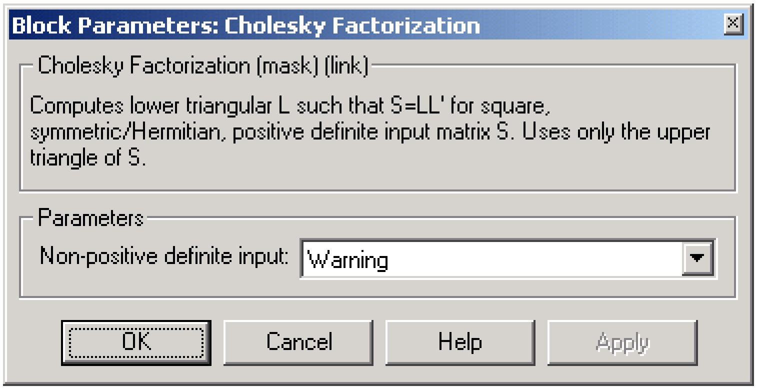
To generate a valid output, the block algorithm requires a positive definite input (see “Input Requirements for Valid Output”). Set the **Non-positive definite input** parameter to determine how the block responds to a non-positive definite input:

- **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 factorization. A partial factorization will be present in the upper left corner of the output.
- **Error** — Display an error dialog and terminate the simulation.

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 non-positive definite matrix inputs: **Ignore**, **Warning**, or **Error**. See “Response to Non-Positive Definite Input”. Tunable.

References

Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

Cholesky Factorization

Supported Data Types

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

To learn how to convert to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

Autocorrelation LPC	DSP Blockset
Cholesky Inverse	DSP Blockset
Cholesky Solver	DSP Blockset
LDL Factorization	DSP Blockset
LU Factorization	DSP Blockset
QR Factorization	DSP Blockset
`chol`	MATLAB

See "Factoring Matrices" on page 6-8 for related information. Also see "Matrix Factorizations" on page 7-10 for a list of all the blocks in the Matrix Factorizations library.

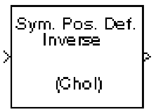
Purpose

Compute the inverse of a 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.

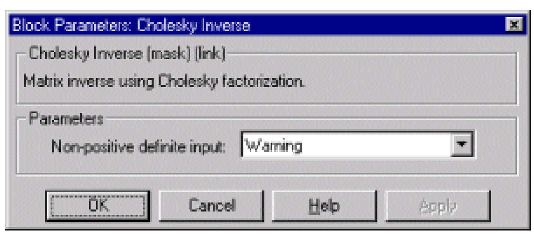
$$S^{-1} = (LL^*)^{-1}$$

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.

Dialog Box



Non-positive definite input

Response to non-positive definite matrix inputs. Tunable.

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 to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

Cholesky Factorization	DSP Blockset
Cholesky Solver	DSP Blockset
LDL Inverse	DSP Blockset
LU Inverse	DSP Blockset
Pseudoinverse	DSP Blockset
`inv`	MATLAB

See "Inverting Matrices" on page 6-10 for related information. Also see "Matrix Inverses" on page 7-11 for a list of all the blocks in the Matrix Inverses library.

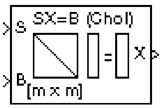
Purpose

Solve the equation S*X*=B for *X* when S is a square Hermitian positive definite matrix.

Library

Math Functions / Matrices and Linear Algebra / Linear System Solvers

Description



The Cholesky Solver block solves the linear system S*X*=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-hand 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.

A length-M vector input for right-hand 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

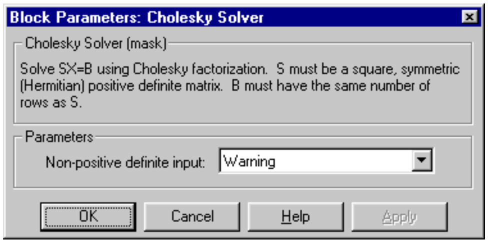
$$LL^*X = B$$

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



Non-positive definite input

Response to non-positive definite matrix inputs. Tunable.

Supported Data Types

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

To learn how to convert to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Autocorrelation LPC	DSP Blockset
Cholesky Factorization	DSP Blockset
Cholesky Inverse	DSP Blockset
LDL Solver	DSP Blockset
LU Solver	DSP Blockset
QR Solver	DSP Blockset
`chol`	MATLAB

See “Solving Linear Systems” on page 6-7 for related information. Also see “Linear System Solvers” on page 7-9 for a list of all the blocks in the Linear System Solvers library.

Purpose

Compute the complex cepstrum of an 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.

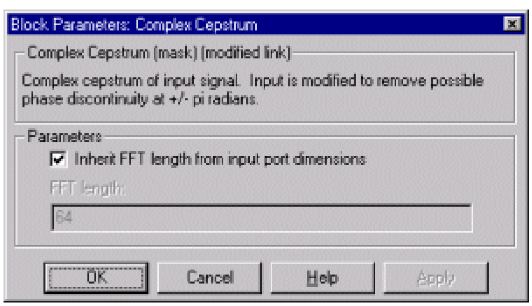
The output is a real M_o-by-N matrix, where M_o is specified by the **FFT length** parameter. Each output column contains the length-M_o complex cepstrum of the corresponding input column.

```
y = cceps(u,Mo)    % Equivalent MATLAB code
```

When the **Inherit FFT length from input port dimensions** check box is selected, 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 (i.e., 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 **Inherit FFT length from input port dimensions** is not selected.

Supported Data Types

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

To learn how to convert to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

DCT	DSP Blockset
FFT	DSP Blockset
Real Cepstrum	DSP Blockset
`cceps`	Signal Processing Toolbox

Also see "Transforms" on page 7-19 for a list of all the blocks in the Transforms library.

Complex Exponential

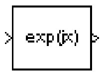
Purpose

Compute the complex exponential function.

Library

Math Functions / Math Operations

Description

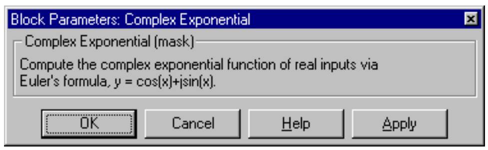


The Complex Exponential block computes the complex exponential function for each element of the real input, u.

$$y = e^{ju} = \cos u + j\sin u$$

where $j = \sqrt{-1}$. The output is complex, with the same size and frame status as the input.

Dialog Box



Supported Data Types

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

To learn how to convert to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

Math Function	Simulink
Sine Wave	DSP Blockset
`exp`	MATLAB

Also see "Math Operations" on page 7-9 for a list of all the blocks in the Math Operations library.

Constant Diagonal Matrix

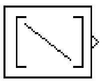
Purpose

Generate a square, diagonal matrix.

Library

- DSP Sources
- Math Functions / Matrices and Linear Algebra / Matrix Operations

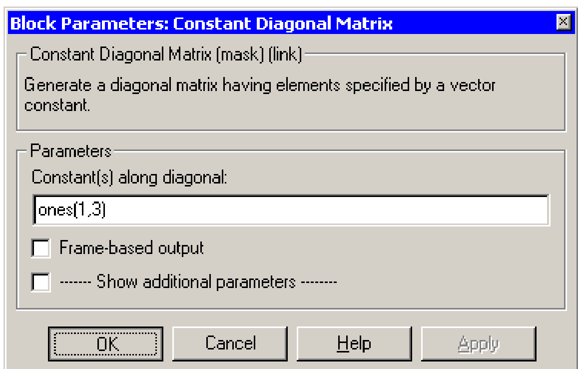
Description



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 the **Frame-based output** check box is selected; otherwise, the output is sample-based.

Dialog Box



Constant(s) along diagonal

Specify the values of the elements along the diagonal. You may input a scalar or a vector. This parameter is tunable.

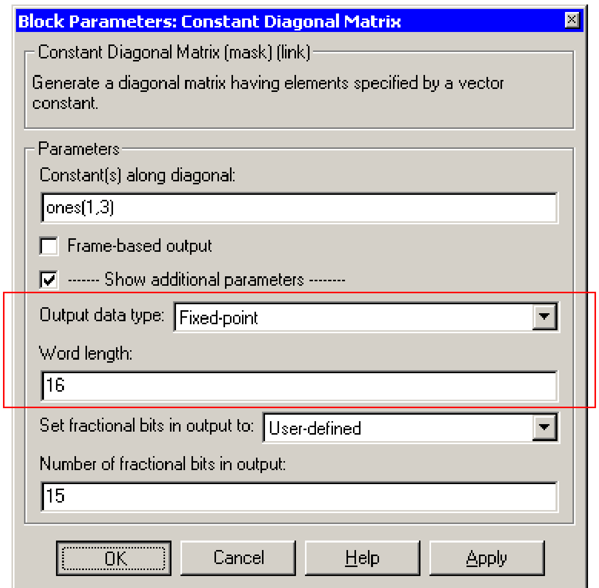
If you specify any data type information in this field, it is overridden by the value of the **Output data type** parameter.

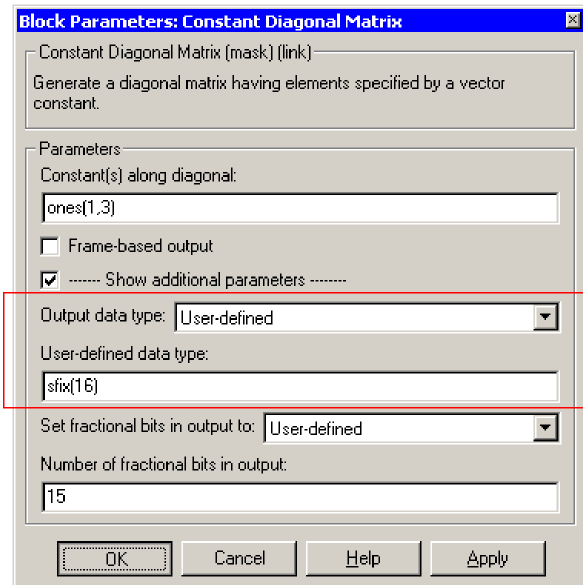
Frame-based output

Select to cause output to be frame-based. Otherwise, output is sample-based.

Show additional parameters

If selected, additional parameters specific to implementation of the block become visible as shown.





Output data type

Specify the output data type in one of the following ways:

- Choose one of the built-in data types from the drop-down list.
- Choose **Fixed-point** to specify the output data type and scaling in the **Word length**, **Set fractional bits in output to,** and **Number of fractional bits in output** parameters.
- Choose **User-defined** to specify the output data type and scaling in the **User-defined data type**, **Set fractional bits in output to,** and **Number of fractional bits in output** 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.

- 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.

Word length

Specify the word length, in bits, of the fixed-point output data type. This parameter is only visible if **Fixed-point** is selected 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 the Fixed-Point Blockset. This parameter is only visible if **User-defined** is selected for the **Output data type** parameter.

Set fractional bits 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 **Number of fractional bits in output** parameter.

This parameter is only visible if **Fixed-point** or **User-defined** is selected for the **Output data type** parameter, and if the specified output data type is a fixed-point data type.

Number of fractional bits in output

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 if **Fixed-point** or **User-defined** is selected for the **Output data type** parameter, and if **User-defined** is selected for the **Set fractional bits in output to** parameter.

Supported Data Types

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

To learn how to convert to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

Create Diagonal Matrix	DSP Blockset
DSP Constant	DSP Blockset
Identity Matrix	DSP Blockset
`diag`	MATLAB

Also see the following topics:

- "Creating Signals Using Constant Blocks" on page 3-33 — How to use this and other blocks to generate constant signals
- "DSP Sources" on page 7-3 — List of all blocks in the DSP Sources library
- "Matrix Operations" on page 7-11 — List of all blocks in the Matrix Operations library.

Constant Ramp

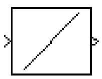
Purpose

Generate a ramp signal with length based on input dimensions.

Library

DSP Sources

Description



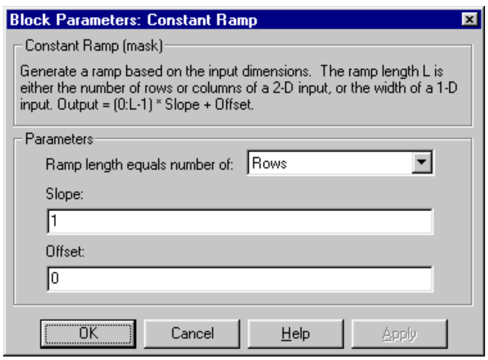
The Constant Ramp block generates the constant ramp signal

```
y = (0:L-1)*m + b
```

where m is the slope specified by the scalar **Slope** parameter, 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



Ramp length equals number of

The dimension of the input matrix that determines the length of the output ramp, **Rows** or **Columns**.

Slope

The slope of the ramp, a scalar.

Offset

The *y*-intercept of the ramp, a scalar.

Supported Data Types

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

To learn how to convert to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Create Diagonal Matrix	DSP Blockset
Identity Matrix	DSP Blockset
DSP Constant	DSP Blockset

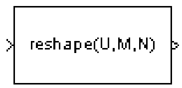
Also see the following topics:

- “Creating Signals Using Constant Blocks” on page 3-33 — How to use this and other blocks to generate constant signals
- “DSP Sources” on page 7-3 — List of all blocks in the DSP Sources library

Purpose Reshape a 1-D or 2-D input to a 2-D matrix with the specified dimensions.

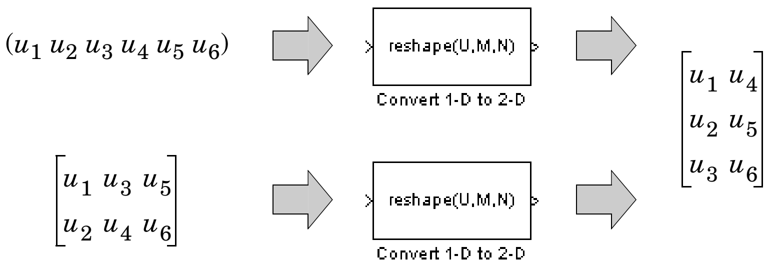
Library Signal Management / Signal Attributes

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
```

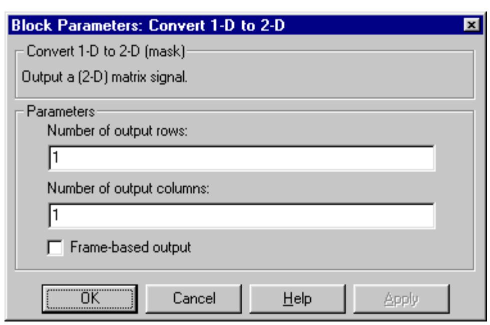
The input is reshaped *columnwise*, as shown in the two cases below. The length-6 vector and the 2-by-3 matrix are both reshaped to the same 3-by-2 output matrix.



An error is generated if $(M_o{*}N_o) \neq (M_i{*}N_i)$. That is, the total number of input elements must be conserved in the output.

The output is frame-based if the **Frame-based output** check box is selected; otherwise, the output is sample-based.

Dialog Box



Number of output rows

The number of rows, M_o, in the output matrix. Tunable.

Number of output columns

The number of rows, N_o, in the output matrix. Tunable.

Frame-based output

Creates a frame-based output when selected. Tunable.

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 to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

Buffer	DSP Blockset
Convert 2-D to 1-D	DSP Blockset
Frame Status Conversion	DSP Blockset
Reshape	Simulink
Submatrix	DSP Blockset

Also see "Signal Attributes" on page 7-15 for a list of all the blocks in the Signal Attributes library.

Convert 2-D to 1-D

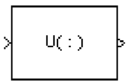
Purpose

Convert a 2-D matrix input to a 1-D vector.

Library

Signal Management / Signal Attributes

Description

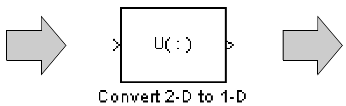


The Convert 2-D to 1-D block reshapes an M-by-N matrix input to a 1-D vector with length M∗N.

```
y = u(:)            % Equivalent MATLAB code
```

The input is reshaped *columnwise*, as shown below for a 3-by-2 matrix.

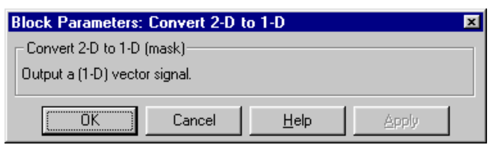
$$\begin{bmatrix} u_1 & u_4 \\ u_2 & u_5 \\ u_3 & u_6 \end{bmatrix}$$



$$(u_1\ u_2\ u_3\ u_4\ u_5\ u_6)$$

The output is always sample-based.

Dialog Box



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 to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Buffer	DSP Blockset
Convert 1-D to 2-D	DSP Blockset
Frame Status Conversion	DSP Blockset
Reshape	Simulink
Submatrix	DSP Blockset

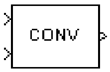
Also see “Signal Attributes” on page 7-15 for a list of all the blocks in the Signal Attributes library.

Convolution

Purpose Compute the convolution of two inputs.

Library Signal Operations

Description



The Convolution block convolves corresponding columns (channels) of an M_u-by-N input matrix u and an M_v-by-N input matrix v. The block does not accept sample-based full-dimension matrix inputs, and the outputs are always sample-based.

Sections of This Reference Page

Convolving Frame-Based Inputs

Matrix inputs must be frame-based. The output, y, is a frame-based (M_u+M_v-1)-by-N matrix whose jth column has elements

$$y_{i(\;),j} = \sum_{k=1}^{max(M_u, M_v)} u_{k,j} v^*_{(i-k+1),j} \qquad 1 \le i \le (M_u + M_v - 1)$$

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 convolved with each channel of the multichannel input. For example, if 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

$$y_{i,j} = \sum_{k=1}^{max(M_u, M_v)} u_k v^*_{(i-k+1),j} \qquad 1 \le i \le (M_u + M_v - 1)$$

Convolving Sample-Based Inputs

If u and v are sample-based vectors with lengths M_u and M_v, the Convolution block performs the vector convolution

$$y_i = \sum_{k=1}^{max(M_u, M_v)} u_k v^*_{(i-k+1)} \qquad 1 \le i \le (M_u + M_v - 1)$$

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.

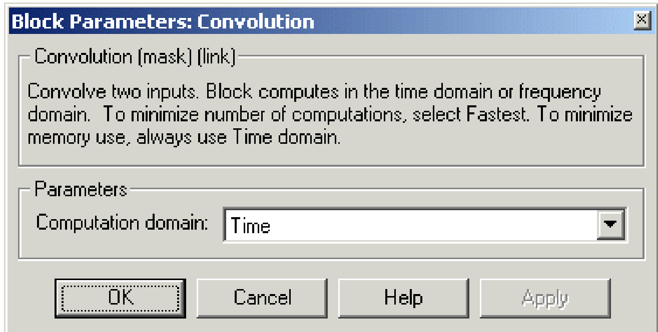
The Convolution block does not accept sample-based full-dimension matrix inputs, or mixed sample-based row vector and column vector inputs.

Setting the Computation Domain Parameter

You must set the domain in which the block computes convolutions in the **Computation domain** parameter. You can select one of the following settings:

- **Time** — The block computes in the time domain, which minimizes memory use.
- **Frequency** — The block computes in the frequency domain, which may require fewer computations than computing in the time domain (depending on the input length).
- **Fastest** — The block computes in the domain that minimizes the number of computations (time domain or frequency domain).

Dialog Box



Computation domain

The domain in which the block computes convolutions: **Time**, **Frequency**, or **Fastest**. To minimize memory use, select **Time**. To minimize the number of computations, select **Fastest**. For more information, see “Setting the Computation Domain Parameter” on page 7-101

Supported Data Types

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

To learn how to convert to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

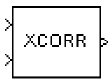
Correlation	DSP Blockset
`conv`	MATLAB

Also see “Signal Operations” on page 7-16 for a list of all the blocks in the Signal Operations library.

Purpose Compute the correlation along the columns of two inputs.

Library Statistics

Description



The Correlation block computes the cross-correlation of corresponding columns (channels) of the M_u-by-N input matrix u and M_v-by-N input matrix v. The frame status of both inputs 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.

Sections of This Reference Page

- "Correlating Frame-Based Inputs" on page 7-103
- "Correlating Sample-Based Inputs" on page 7-104
- "Setting the Computation Domain Parameter" on page 7-104
- "Dialog Box" on page 7-105
- "Supported Data Types" on page 7-102
- "See Also" on page 7-105

Correlating Frame-Based Inputs

Matrix inputs 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, if 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

$$y_{(i+M_v),j} = \sum_{k=1}^{max(M_u, M_v)} u_k v^*_{(k-i),j} \qquad -M_u < i < M_v$$

Correlating Sample-Based Inputs

Matrix inputs cannot be sample based, so all sample-based inputs are column vectors or 1-D vectors. (the block does not support 2-D row vector inputs.) If u and v are sample-based vectors with lengths M_u and M_v, the Correlation block performs the vector cross-correlation

$$y_{(i+M_v)} = \sum_{k=1}^{max(M_u, M_v)} u_k v^*_{(k-i)} \qquad -M_u < i < M_v$$

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.

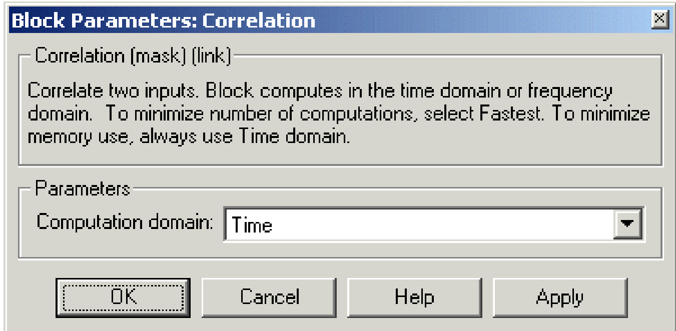
The Correlation block does not accept sample-based full-dimension matrix inputs or 2-D row vector inputs.

Setting the Computation Domain Parameter

You must set the domain in which the block computes correlations in the **Computation domain** parameter. You can select one of the following settings:

- **Time** — The block computes in the time domain, which minimizes memory use.
- **Frequency** — The block computes in the frequency domain, which may require fewer computations than computing in the time domain (depending on the input length).
- **Fastest** — The block computes in the domain that minimizes the number of computations (time domain or frequency domain).

Dialog Box



Computation domain

The domain in which the block computes correlations: **Time**, **Frequency**, or **Fastest**. To minimize memory use, select **Time**. To minimize the number of computations, select **Fastest**. For more information, see “Setting the Computation Domain Parameter” on page 7-104.

Supported Data Types

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

To learn how to convert to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Autocorrelation	DSP Blockset
Convolution	DSP Blockset
`xcorr`	Signal Processing Toolbox

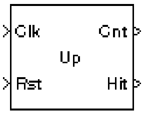
Also see “Statistics” on page 7-18 for a list of all the blocks in the Statistics library.

Counter

Purpose Count up or down through a 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 (i.e., single channel). If both inputs are sample-based, they must have the same sample period. If the `Clk` input is frame-based, the frame period must equal the sample period of the `Rst` input.

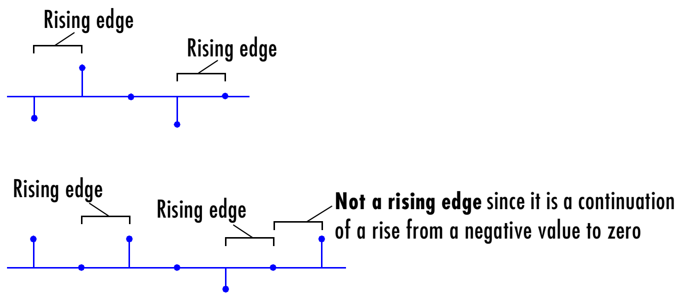
Sections of This Reference Page

Setting the Count Event Parameter

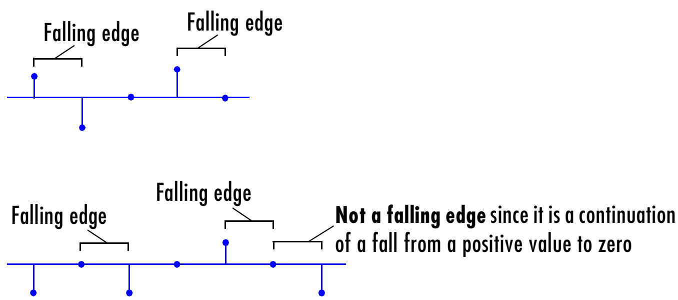
The trigger event for both inputs is specified by the **Count event** parameter, and can be one of the following:

- **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 7-110 below). 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 and clock 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 3-91 and the topic on the **Simulation Parameters** dialog box in the Simulink 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 (e.g., 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 (e.g., 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 (i.e., 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

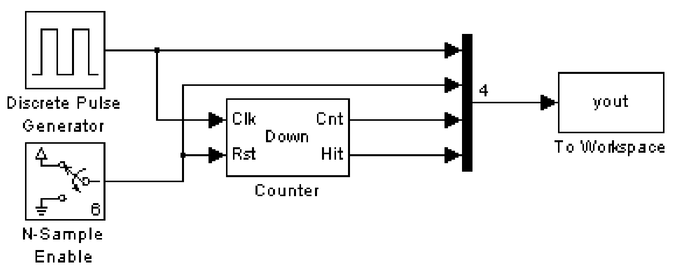
The block operates in free-running mode when **Free running** is selected from the **Count event** menu.

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 $M*T_s$.

Example

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.



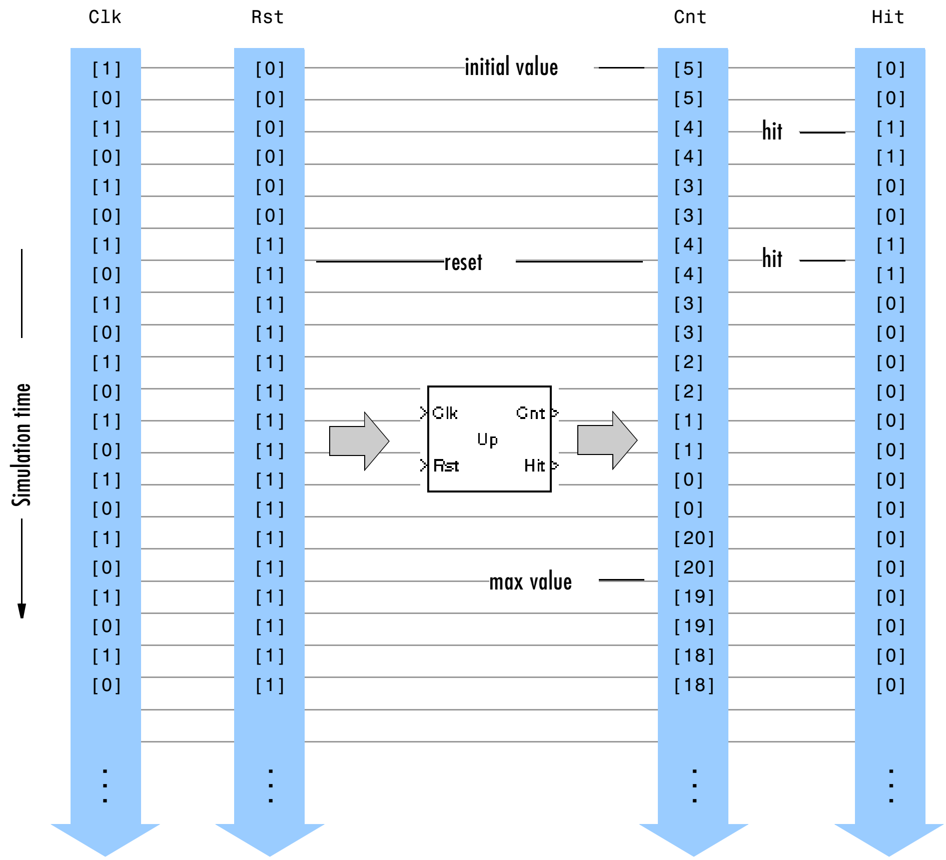
To run the model, first select **Simulation Parameters** from the **Simulation** menu, and set the **Stop time** to `30`. Then adjust the block parameters as described below. (Use the default settings for the Pulse Generator and To Workspace blocks.)

- 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** ☑

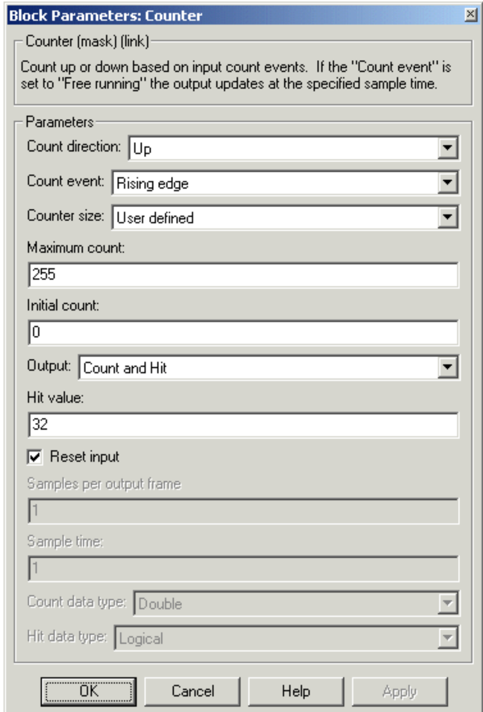
• Set the **Number of inputs** parameter of the Mux block to `4`.

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.



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 7-106.

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 7-108.

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 **Free running** is selected in the **Count event** menu.

Sample time

The output sample period, T_s, in free-running mode. This parameter is available when **Free running** is selected in the **Count event** menu.

Count data type

The data type of the output from the `Cnt` output port. This parameter is available when **Free running** is selected in the **Count event** menu.

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 A-11. This parameter is available when **Free running** is selected in the **Count event** menu and **Hit** or **Count and Hit** is selected in the **Output** menu.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Boolean — The block accepts Boolean inputs to the `Rst` port. The block may 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 A-11. To learn how to disable Boolean output support, see "Steps to Disabling Boolean Support" on page A-12.

To learn how to convert to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

Edge Detector	DSP Blockset
N-Sample Enable	DSP Blockset
N-Sample Switch	DSP Blockset

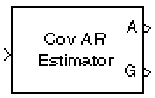
Also see "Switches and Counters" on page 7-15 for a list of all the blocks in the Switches and Counters library.

Covariance AR Estimator

Purpose Compute an estimate of AR model parameters using the 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.

$$H(z) = \frac{G}{A(z)} = \frac{G}{1 + a(2)z^{-1} + \ldots + a(p+1)z^{-p}}$$

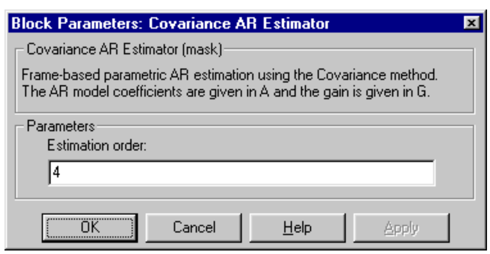
The order, p, of the all-pole model is specified by the **Estimation order** parameter. To guarantee a nonsingular output, you must set the value of p to be less than the input length. Otherwise, the output may be singular.

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`).

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 may 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

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

To learn how to convert to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

Burg AR Estimator	DSP Blockset
Covariance Method	DSP Blockset
Modified Covariance AR Estimator	DSP Blockset
Yule-Walker AR Estimator	DSP Blockset
`arcov`	Signal Processing Toolbox

Also see "Parametric Estimation" on page 7-5 for a list of all the blocks in the Parametric Estimation library.

Covariance Method

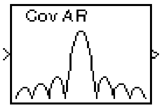
Purpose

Compute a parametric spectral estimate using the covariance method.

Library

Estimation / Power Spectrum Estimation

Description



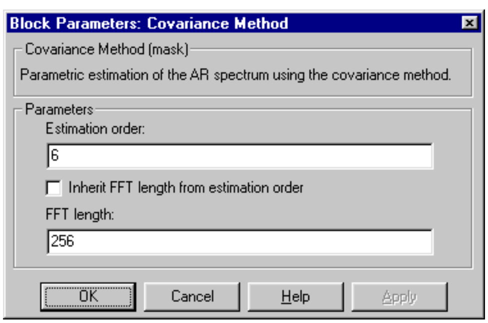
The Covariance Method block estimates the power spectral density (PSD) of the input using the covariance method. This method fits an autoregressive (AR) model to the signal by minimizing the forward prediction error in the least-squares sense. The order of the all-pole model is the value specified by the **Estimation order** parameter, and the spectrum is computed from the FFT of the estimated AR model parameters. To guarantee a nonsingular output, you must set the value of the **Estimation order** parameter to be less than the input length. Otherwise, the output may be singular.

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 **Inherit FFT length from input dimensions** is selected, N_{fft} is specified by the frame size of the input, which must be a power of 2. When **Inherit FFT length from input dimensions** is *not* selected, 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 may 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. If N_{fft} exceeds the input frame size, the frame is zero-padded as needed. This parameter is enabled when **Inherit FFT length from input dimensions** is not selected.

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

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

To learn how to convert to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

Burg Method	DSP Blockset
Covariance AR Estimator	DSP Blockset
Short-Time FFT	DSP Blockset
Modified Covariance Method	DSP Blockset
Yule-Walker Method	DSP Blockset
`pcov`	Signal Processing Toolbox

See "Power Spectrum Estimation" on page 6-6 for related information. Also see a list of all blocks in the Power Spectrum Estimation library.

Create Diagonal Matrix

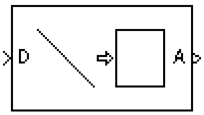
Purpose

Create a square diagonal matrix from diagonal elements.

Library

Math Functions / Matrices and Linear Algebra / Matrix Operations

Description

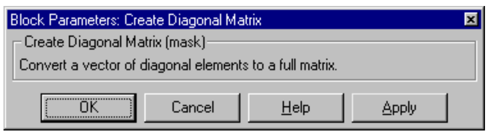


The Create Diagonal Matrix block populates the diagonal of the M-by-M matrix output with the elements contained in the length-M vector input, `D`. The elements off the diagonal are zero.

```
A = diag(D)        Equivalent MATLAB code
```

The output is always sample-based.

Dialog Box



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 to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Constant Diagonal Matrix	DSP Blockset
Extract Diagonal	DSP Blockset
`diag`	MATLAB

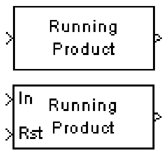
Also see “Matrix Operations” on page 7-11 for a list of all the blocks in the Matrix Operations library.

Cumulative Product

Purpose Compute the 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 size, dimension, rate, frame status, data type, and complexity as the input.

Sections of This Reference Page

Input and Output Characteristics

Valid Input to Multiply. 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.

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 size, dimension, 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"
- "Multiplying Along Channels of Sample-Based Inputs"
- "Resetting the Cumulative Product Along Channels"

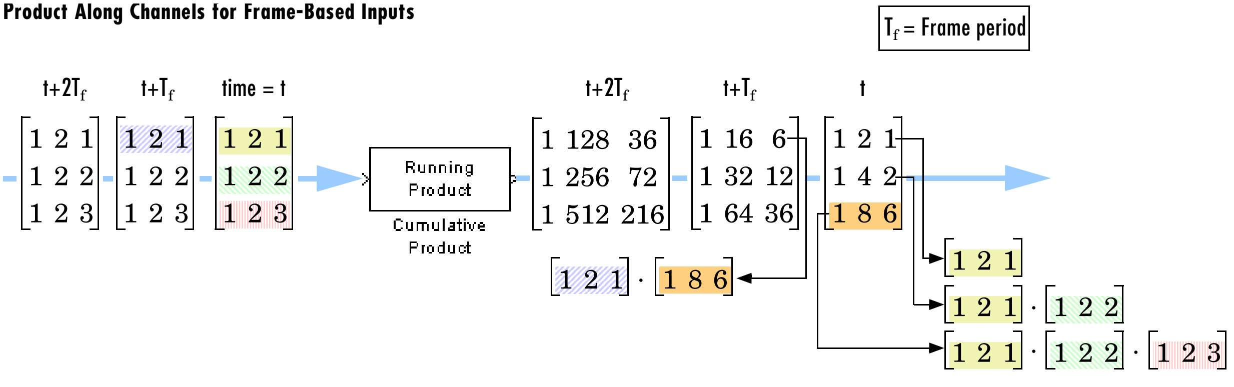
Multiplying Along Channels of Frame-Based Inputs. For frame-based inputs, the block treats each input column as an independent channel. As the following figure and equation illustrate, the output has the following characteristics:

- 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:

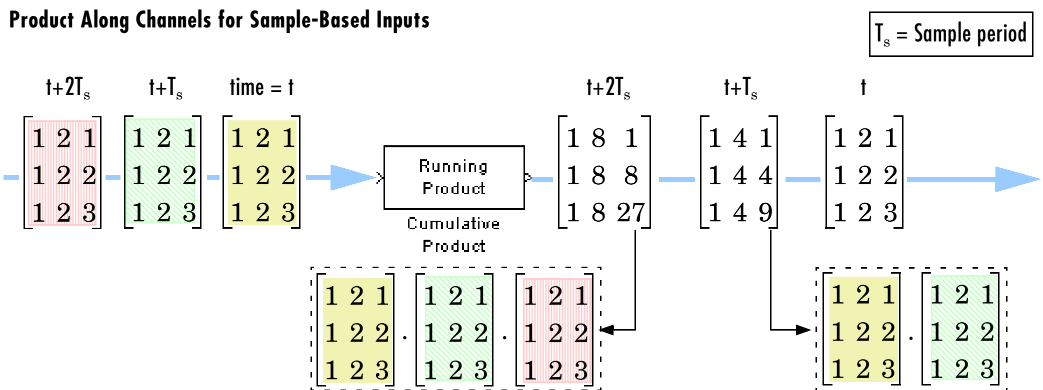
- The first output is the same as the first input.
- Each subsequent output is the element-wise product of the current input (time t) and the previous output (time $t - T_s$, where T_s is the sample period).
- The output has the same size, dimension, frame status, data type, and complexity as the input.

Given an M-by-N sample-based input, u, the output, y, is a sample-based M-by-N matrix with the elements

$$y_{i,j}(t) = u_{i,j}(t) \cdot y_{i,j}(t - T_s) \qquad \begin{array}{l} 1 \le i \le M \\ 1 \le j \le N \end{array}$$

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

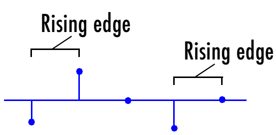


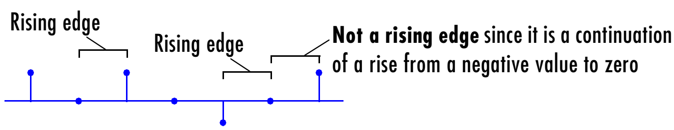
Resetting the Cumulative Product Along Channels. When you set the **Multiply input along** parameter to **Channels (running product)**, you can set the block to reset the running product whenever it detects a reset event at the optional `Rst` port. The rate of the reset signal must be a positive integer 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.

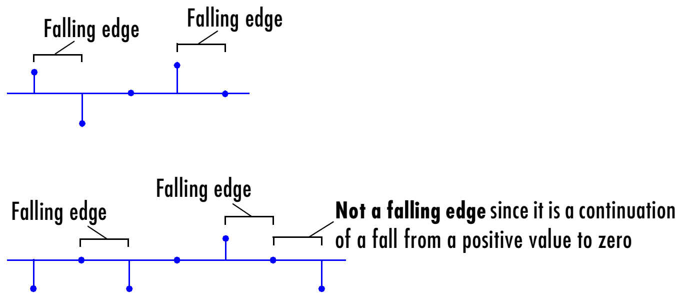
The **Reset port** parameter specifies the reset event, which can be one of the following:

- **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.

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 3-91 and the topic on the **Simulation Parameters** dialog box in the Simulink documentation.

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.

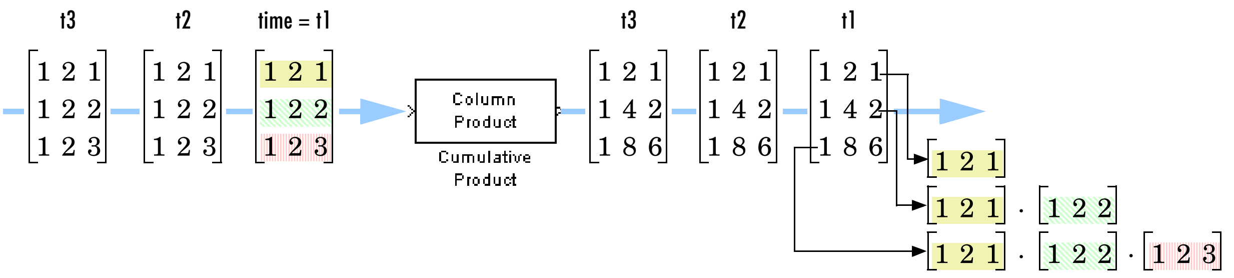
Cumulative Product

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
```

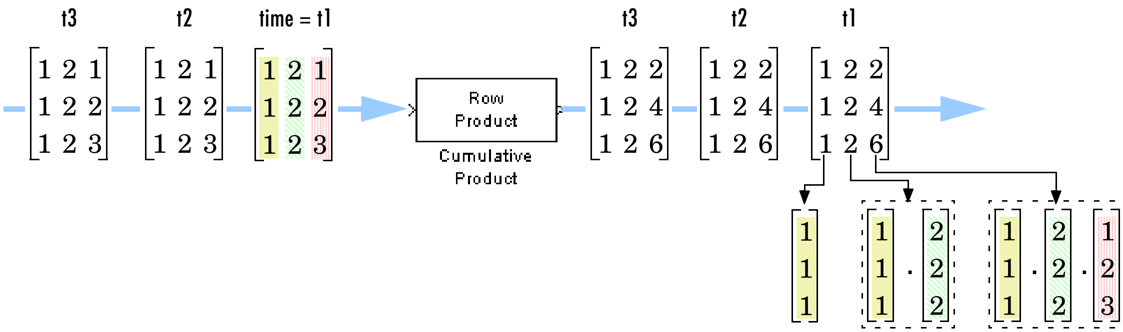
The output has the same size, dimension, frame status, and data type as the input. The nth output column is the element-wise product of the first n input columns.

Given an M-by-N input, u, the output, y, is an M-by-N matrix whose ith row has elements

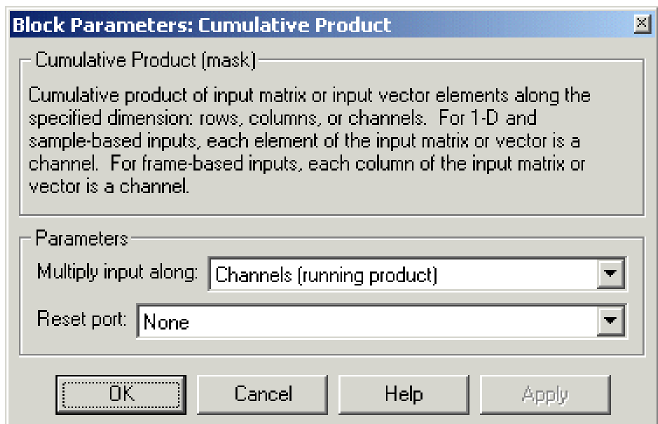
$$y_{i,j} = \prod_{k=1}^{j} u_{i,k} \qquad 1 \le j \le N$$

The block treats length-N 1-D vector inputs as 1-by-N row vectors when multiplying along rows.

Product Along Rows



Dialog Box



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 7-122
- “Multiplying Along Columns” on page 7-125
- “Multiplying Along Rows” on page 7-126

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 7-124.

Supported Data Types

Input and Output Ports	Supported Data Types
Data input port, `In`	• Double-precision floating point • Single-precision floating point
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 to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

Cumulative Sum	DSP Blockset
Matrix Product	DSP Blockset
`cumprod`	MATLAB

Also see "Math Operations" on page 7-9 for a list of all the blocks in the Math Operations library.

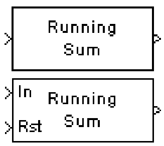
Purpose

Compute the 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 size, dimension, rate, frame status, data type, and complexity as the input.

Sections of This Reference Page

Input and Output Characteristics

Valid Input to Sum. 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.

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 size, dimension, 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 7-130
- "Summing Along Channels of Sample-Based Inputs" on page 7-131
- "Resetting the Cumulative Sum Along Channels" on page 7-131

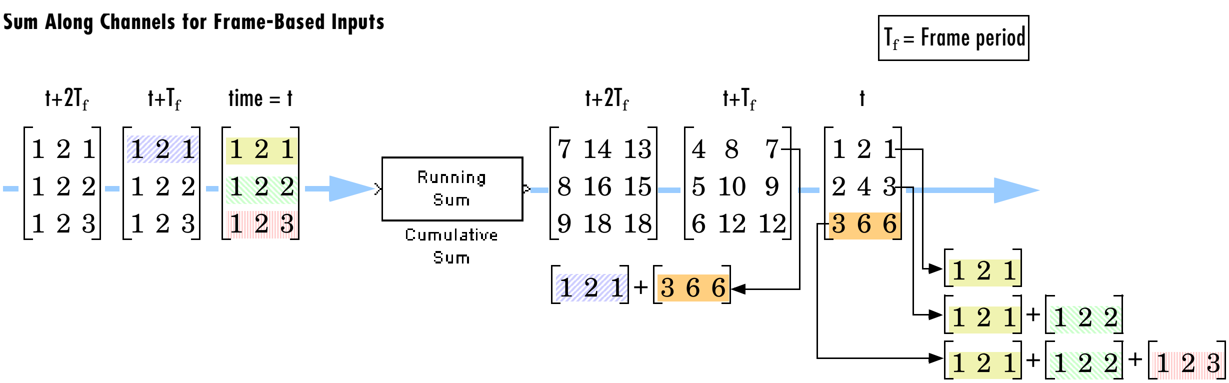
Summing Along Channels of Frame-Based Inputs. For frame-based inputs, the block treats each input column as an independent channel. As the following figure and equation illustrate, the output has the following characteristics:

- 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



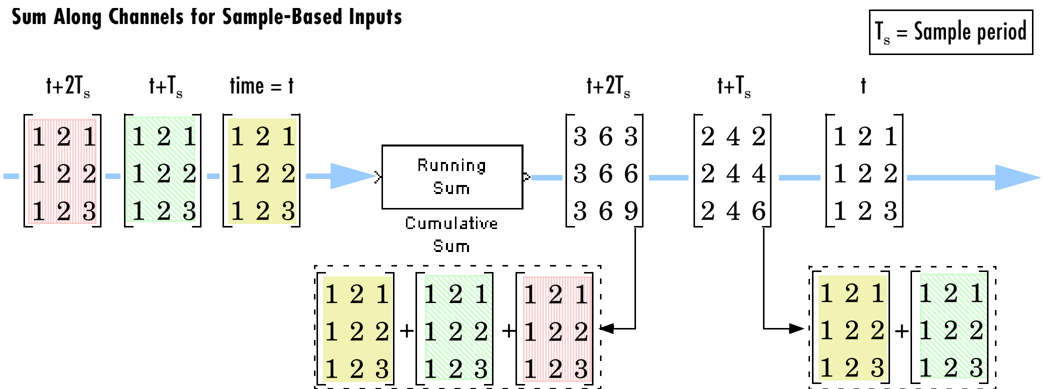
Summing Along Channels of Sample-Based Inputs. For sample-based inputs, the block treats each element of the input matrix as an independent channel. As the following figure and equation illustrate, the output has the following characteristics:

- 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

$$y_{i,j}(t) = u_{i,j}(t) + y_{i,j}(t - T_s) \qquad \begin{array}{l} 1 \le i \le M \\ 1 \le j \le N \end{array}$$

Sum Along Channels for Sample-Based Inputs

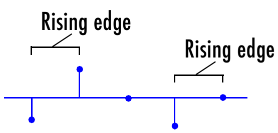


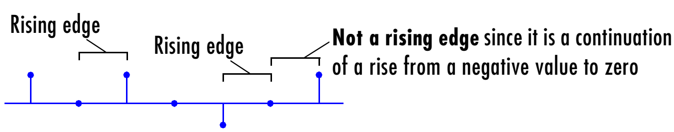
Resetting the Cumulative Sum Along Channels. When you set the **Sum input along** parameter to **Channels (running sum)**, you can set the block to reset the running sum whenever it detects a reset event at the optional `Rst` port. The rate of the reset signal must be a positive integer multiple of the 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.

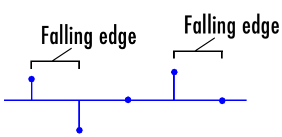
The **Reset port** parameter specifies the reset event, which can be one of the following:

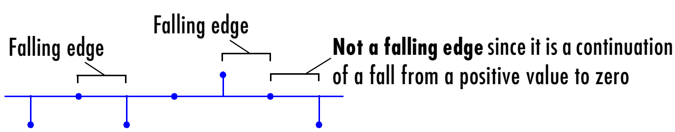
- **None** disables the `Rst` port.
- **Rising edge** — Triggers a reset operation when the `Rst` input does one of the following:
  - Rises from a negative value to a positive value or zero
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value 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.

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 3-91 and the topic on the **Simulation Parameters** dialog box in the Simulink documentation.

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.

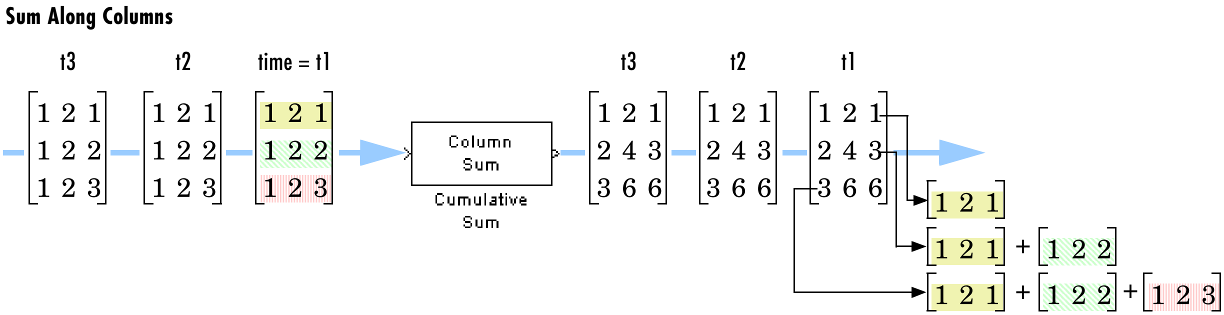
```
y = cumsum(u)          % Equivalent MATLAB code
```

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.



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
```

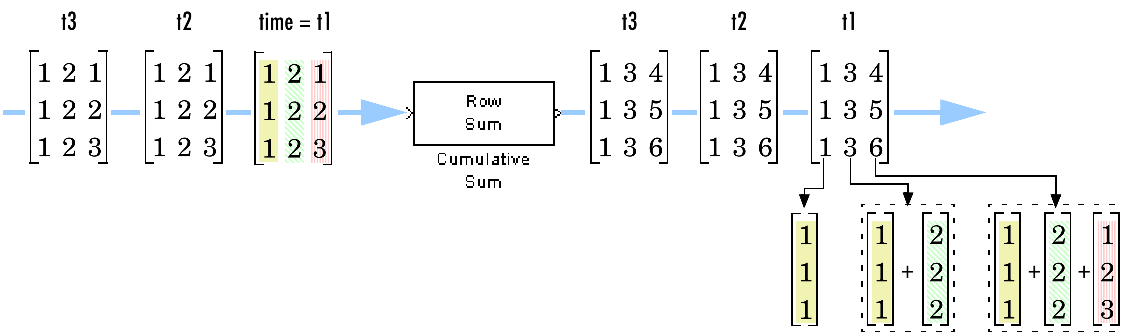
The output has the same size, dimension, frame status, and data type as the input. The nth output column is the sum of the first n input columns.

Given an M-by-N input, u, the output, y, is an M-by-N matrix whose ith row has elements

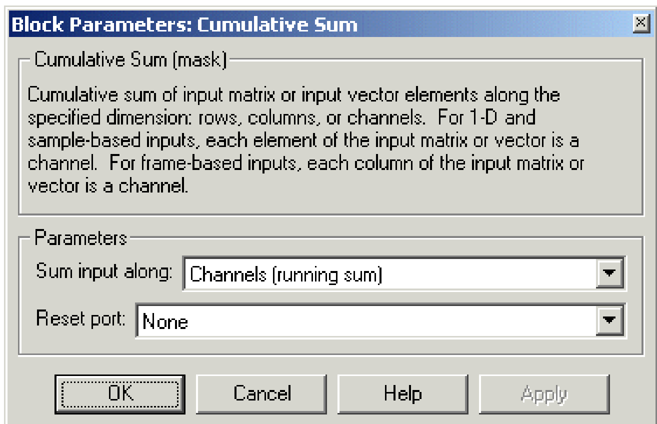
$$y_{i,j} = \sum_{k=1}^{j} u_{i,k} \qquad 1 \le j \le N$$

The block treats length-N 1-D vector inputs as 1-by-N row vectors when summing along rows.

Sum Along Rows



Dialog Box



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 7-130
- “Summing Along Columns” on page 7-133
- “Summing Along Rows” on page 7-134

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 7-131.

Supported Data Types

Input and Output Ports	Supported Data Types
Data input port, `In`	• Double-precision floating point • Single-precision floating point
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 to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

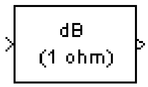
See Also

Cumulative Product	DSP Blockset
Difference	DSP Blockset
Matrix Sum	DSP Blockset
`cumsum`	MATLAB

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

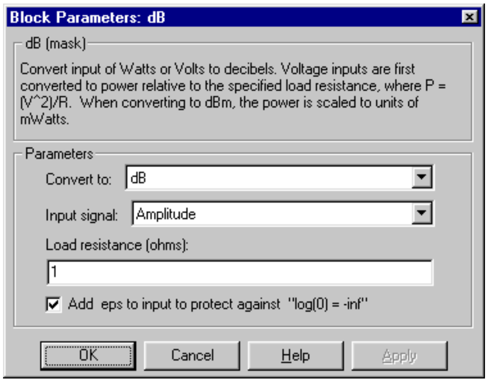
```
y = 10*log10(abs(u)^2/R)
```

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**. Tunable.

Load resistance

The scale factor used to convert voltage inputs to units of power. Tunable.

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 to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

dB Gain	DSP Blockset
Math Function	Simulink
`log10`	MATLAB

Also see “Math Operations” on page 7-9 for a list of all the blocks in the Math Operations library.

Purpose

Apply a gain specified in decibels.

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.

$$y_{ij} = 10u_{ij}^{(g_{ij}/k)}$$

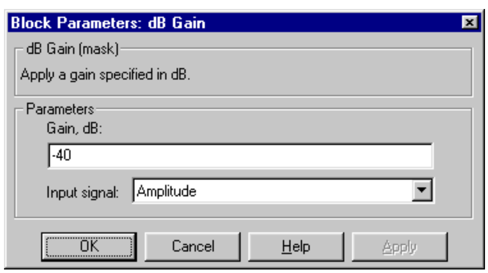
The value of k is 10 for power signals (select **Power** as the **Input signal** parameter) and 20 for voltage signals (select **Amplitude** as the **Input signal** parameter).

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.

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

To learn how to convert to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

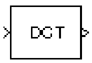
dB Conversion	DSP Blockset
Math Function	Simulink
`log10`	MATLAB

Also see “Math Operations” on page 7-9 for a list of all the blocks in the Math Operations library.

Purpose Compute the DCT of the 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`.




```
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 lth column contains the length-M DCT of the corresponding input column.

$$y(k,l) = w(k)\sum_{m=1}^{M} u(m,l)\cos\frac{\pi(2m-1)(k-1)}{2M}, \qquad k = 1, \ldots, M$$

where

$$w(k) = \begin{cases} \dfrac{1}{\sqrt{M}}, & k = 1 \\ \sqrt{\dfrac{2}{M}}, & 2 \le k \le M \end{cases}$$

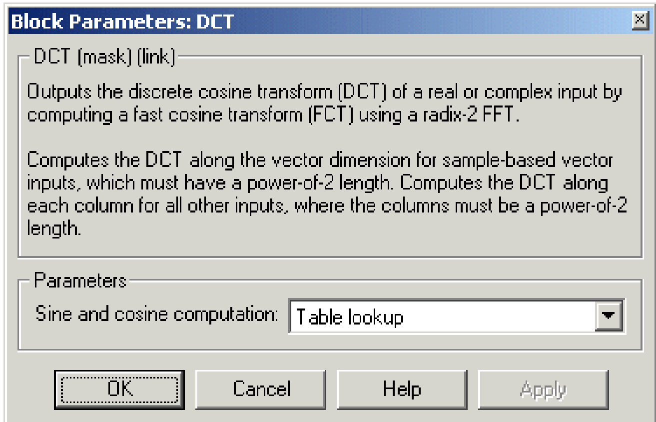
The output is always sample-based, and the output port rate and data type (real/complex) are the same as those of the input port.

For convenience, length-M 1-D vector inputs and *sample-based* length-M row vector inputs are processed as single channels (i.e., 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 in the FFT and fast DCT algorithms used to compute the DCT. 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.

Dialog Box



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.

Supported Data Types

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

To learn how to convert to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

Complex Cepstrum	DSP Blockset
FFT	DSP Blockset
IDCT	DSP Blockset
Real Cepstrum	DSP Blockset
`dct`	Signal Processing Toolbox

Also see "Transforms" on page 7-19 for a list of all the blocks in the Transforms library.

Delay Line

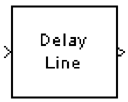
Purpose

Rebuffer a sequence of inputs with a one-sample shift.

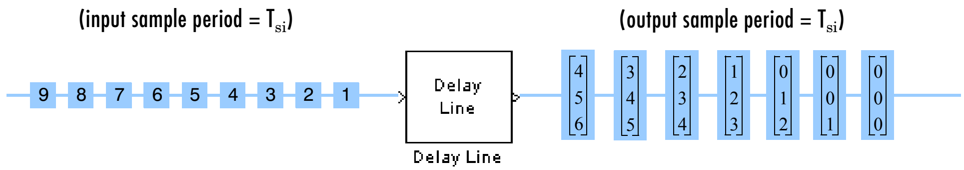
Library

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.

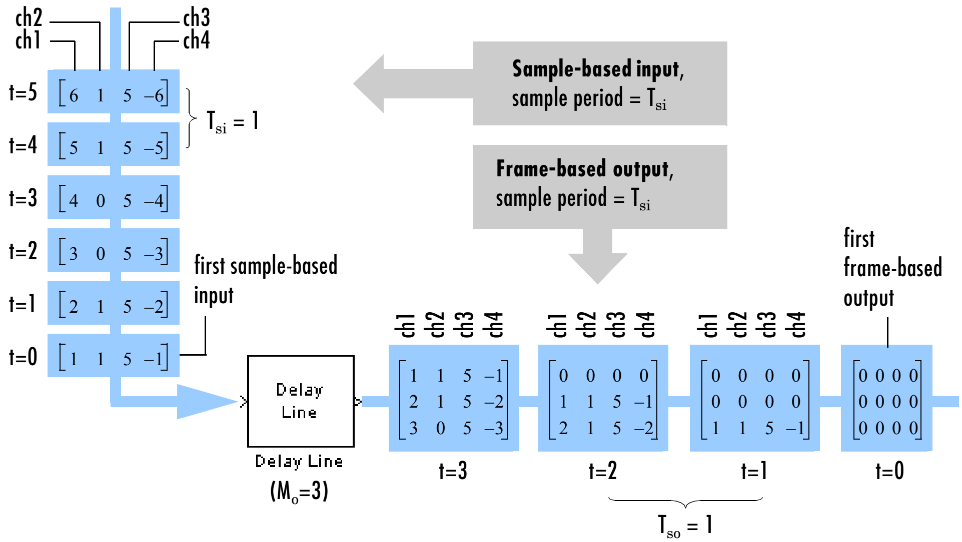
Sample-Based Operation

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" below). Sample-based full-dimension matrix inputs are not accepted.

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 model below, 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" below). 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

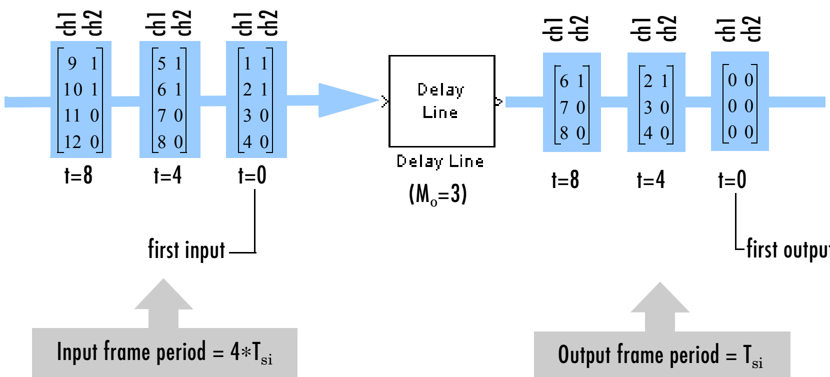
In frame-based operation, the Delay Line block rebuffers a sequence of frame-based M_i-by-N matrix inputs into a sequence of frame-based M_o-by-N matrix outputs, where M_o is the output frame size specified by the **Delay line size** parameter. Depending on whether M_o is greater than, less than, or 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" below).

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 model below, the block rebuffers a two-channel frame-based input with a **Delay line size** of 3.

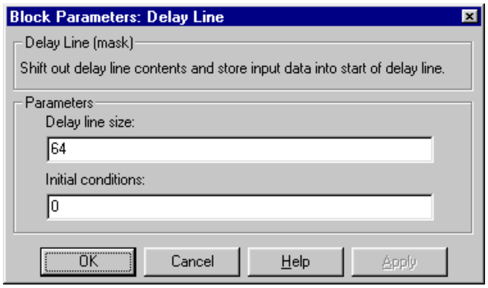


The first output frame in the example is a product of the latency of the Delay Line block; it is all zeros because the **Initial conditions** is set to be zero. Since the input frame size, 4, is larger than the output frame size, 3, only the last three samples in each input frame are propagated to the 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). If 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. If 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.

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 to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

Buffer	DSP Blockset
Triggered Delay Line	DSP Blockset

See "Buffering Sample-Based and Frame-Based Signals" on page 3-47 for related information. Also see "Buffers" on page 7-14 for a list of all the blocks in the Buffers library.

Detrend

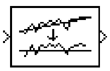
Purpose

Remove a linear trend from a vector.

Library

Statistics

Description



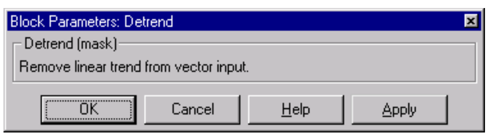
The Detrend block removes a linear trend from the length-M input vector, u, by subtracting the straight line that best fits the data in the least-squares sense.

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 ith element in the input vector. The output, $y = u - \hat{u}$, 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 to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Cumulative Sum	DSP Blockset
Difference	DSP Blockset
Least Squares Polynomial Fit	DSP Blockset
Unwrap	DSP Blockset
`detrend`	MATLAB

Also see “Statistics” on page 7-18 for a list of all the blocks in the Statistics library.

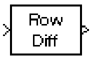
Purpose

Compute the 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 jth column has elements

$$y_{i,j} = u_{i+1,j} - u_{i,j} \qquad 1 \le i \le (M-1)$$

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 frame-based inputs, the output is a frame-based M-by-N matrix whose jth column has elements

$$y_{i,j} = u_{i+1,j} - u_{i,j} \qquad 2 \le i \le M$$

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).

$$y_{1,j}(t) = u_{M,j}(t - T_f) - u_{1,j}(t)$$

Rowwise Differencing

When the **Difference along** parameter is set to **Rows**, the block computes differences between adjacent row elements.

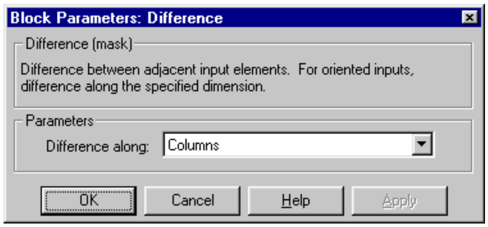
```
y = diff(u,[],2)        % Equivalent MATLAB code
```

The output is an M-by-(N-1) matrix whose ith row has elements

$$y_{i,j} = u_{i,j+1} - u_{i,j} \qquad 1 \le j \le (N-1)$$

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.

Dialog Box



Difference along

The dimension along which to compute element-to-element differences. **Columns** specifies columnwise differencing, while **Rows** specifies rowwise differencing. Tunable.

Supported Data Types

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

To learn how to convert to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Cumulative Sum	DSP Blockset
`diff`	MATLAB

Also see “Math Operations” on page 7-9 for a list of all the blocks in the Math Operations library.

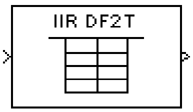
Purpose

Independently filter each channel of the input over time using a specified time-varying or static digital filter implementation

Library

Filtering / Filter Designs

Description



Note Use this block to efficiently implement floating-point filters that you have already designed. The following 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 exact filter implementation as the Digital Filter 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 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.)

The block filters each channel of the input signal independently over time. The output size, frame status, dimension, and data type 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.

The outputs of the block numerically match the outputs of the Digital Filter Design block, the `filter` function in the Signal Processing Toolbox, and the `filter` function in Filter Design Toolbox.

Sections of This Reference Page

- “Supported Filter Structures” on page 7-152
- “Specifying Static Filters” on page 7-153
- “Specifying Time-Varying Filters” on page 7-154

Supported Filter Structures

The selection of filter structures offered in the **Filter structure** parameter depends on whether you set the filter to be IIR with poles and zeros, IIR with all poles, or FIR with all zeros, as summarized in the following table. 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 7-153 and "Specifying Time-Varying Filters" on page 7-154.

Table 7-3: 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 II transposed (second-order sections)	M-by-6 second-order section (SOS) matrix. See "Specifying the SOS Matrix (Biquadratic Filter Coefficients)" on page 7-159.
IIR (all poles)	Direct form Transposed direct form	Denominator coefficients vector [a0, a1, a2, ..., am]
	Lattice AR	Reflection coefficients vector [k1, k2, ..., kn]

Table 7-3: Filter Structures and Filter Coefficients (Continued)

Transfer Function Type	Supported Filter Structures	Filter Coefficient Specification
FIR (all zeros)	Direct form Transposed direct form	Numerator coefficients vector [b0, b1, b2, ..., bn]
	Lattice MA	Reflection coefficients vector [k1, k2, ..., kn]

Specifying Static Filters

To specify a *static filter* whose coefficients are fixed in time, set the **Coefficient source** parameter to **Dialog parameter(s)**. 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 Table 7-3, Filter Structures and Filter Coefficients:

- **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 7-159.

Tuning the Filter Coefficient Values During Simulation. You can change the value of the static filter coefficients during a running simulation. To tune the coefficients during a simulation, double-click the block, type in the new vector(s) of filter coefficients, and click **Apply**. You cannot change the filter order, so you cannot change the number of elements in the vector(s) of filter coefficients.

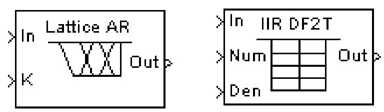
Specifying Time-Varying Filters

Note The block does not support time-varying biquadratic direct form II transposed 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. You can filter multiple channels with each filter, but you cannot apply different filters to each channel; all channels must be filtered with the same filter.

To specify a time-varying filter, you must do the following:

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 7-155.
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 sometimes *must be of certain lengths*. To learn more, see "Providing Filter Coefficient Vectors at Block Input Ports" on page 7-156.
4. Select or clear the **First denominator coefficient = 1, remove 1/a0 term in the structure** parameter depending on whether your first denominator coefficient is always 1. To learn more, see "Removing the 1/a0 Term in the Filter Structure" on page 7-158.

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 *multiple filters*: one filter for each sample in the current frame.

Though the following figure shows the block filtering one channel, the block can filter multiple channels. (The block *can* apply a single filter to multiple channels, but *cannot* apply a different filter to each channel.) To learn how to correctly specify vectors of time-varying filter coefficients, see “Providing Filter Coefficient Vectors at Block Input Ports” on page 7-156.

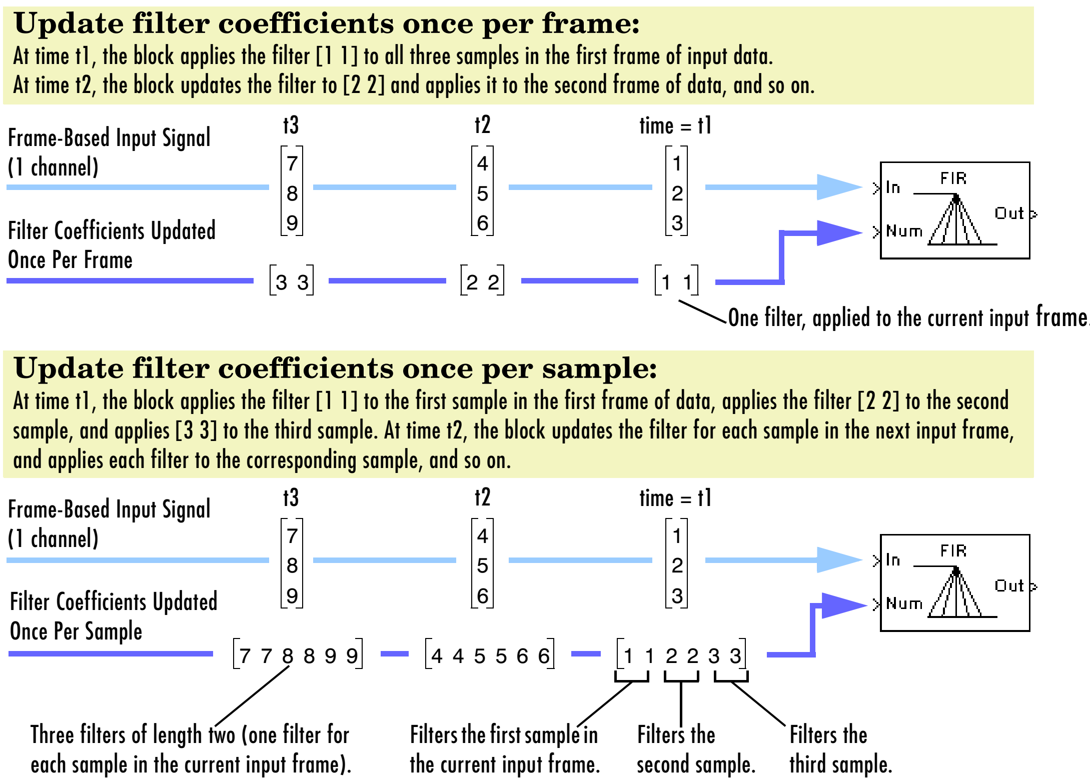


Figure 7-1: Options for Updating a Time-Varying Filter for a Frame-Based Input Signal

Providing Filter Coefficient Vectors at Block Input Ports. As illustrated in Figure 7-1, 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 — Table 7-4, Length Requirements for Time-Varying Filter Coefficient Vectors, on page 7-156
- Rate requirements — Table 7-5, Rate Requirements for Time-Varying Filter Coefficient Vectors, on page 7-157

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

Table 7-4: Length Requirements for Time-Varying Filter Coefficient Vectors

Coefficient Update Rate	**How to Specify Filter Coefficient Vectors** (Also see Figure 7-1)	**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, but represents multiple filters of the same length: 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 frame of three samples: $$\left[b_0\ b_1\ B_0\ B_1\ \beta_0\ \beta_1\right]$$ 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.)

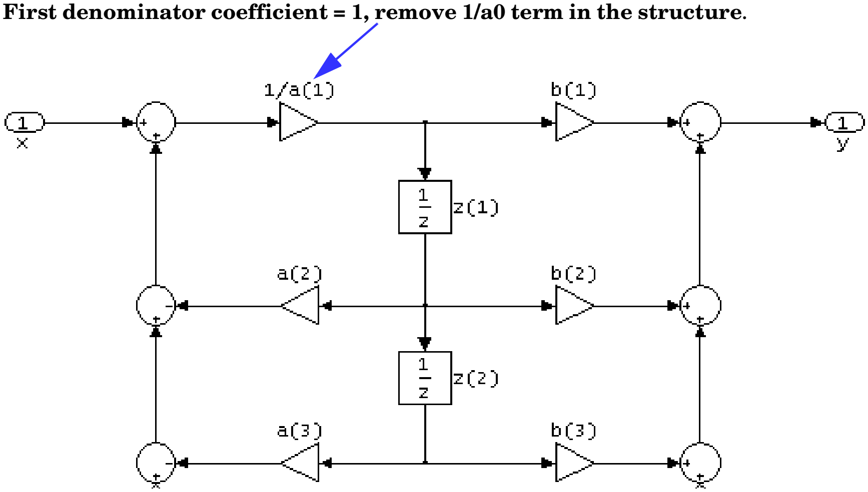
The time-varying filter coefficient vectors can be sample- or frame-based row or column vectors. The vector of filter coefficients must arrive at their input ports at the same times that the frames of input data arrive at their input port, as indicated in the following table.

Table 7-5: Rate Requirements for Time-Varying Filter Coefficient Vectors

Input Signal	**Time-Varying Filter Coefficient Vectors**	**Rate Requirements** (Also see Figure 7-1)
Sample-based	Sample-based	Sample rates of input and filter coefficients must be equal
Sample-based	Frame-based	Input sample rate must equal filter coefficient frame rate
Frame-based	Sample-based	Input frame rate must equal filter coefficient sample rate
Frame-based	Frame-based	Frame rates of input and filter coefficients must be equal

Removing the 1/a0 Term in the Filter Structure. If 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 1/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.

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



Specifying the SOS Matrix (Biquadratic Filter Coefficients)

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

- **Transfer function type — IIR (poles & zeros)**
- **Filter structure — Biquadratic direct form II transposed (sos)**
- **SOS matrix (Mx6)** — M-by-6 *SOS matrix* (described in detail below).

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. 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.

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

Note The block uses a value of 1 for the zero-delay denominator coefficients (a_{11} to a_{1M}) regardless of the value specified in the **SOS matrix (Mx6)** parameter.

Specifying Initial Conditions

By default, the block initializes the internal filter states to zero, 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 Table 7-6, Valid Initial Conditions and Table 7-7, Number of Delay Elements (Filter States). The **Initial conditions** parameter may take one of four forms as described in the following table.

Table 7-6: 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 Table 7-7).
Vector or Matrix (for applying different delay elements to each channel)	For a 3-channel input signal and a filter with two delay elements: $[\mathrm{d}_1\ \mathrm{d}_2\ \mathrm{D}_1\ \mathrm{D}_2\ d_1\ d_2]$ or $\begin{bmatrix} \mathrm{d}_1 & \mathrm{D}_1 & d_1 \\ \mathrm{d}_2 & \mathrm{D}_2 & d_2 \end{bmatrix}$ • 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 Table 7-7). • The matrix must have the same number of rows as the number of delay elements in the filter (specified in Table 7-7), 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.

Table 7-7: Number of Delay Elements (Filter States)

Filter Structure	Number of Delay Elements
Direct form	`#_of_filter_coeffs   1`
Direct form I	• `#_of_zeros   1` • `#_of_poles   1`
Direct form II	`max(#_of_zeros, #_of_poles)   1`
Transposed direct form	`#_of_filter_coeffs   1`
Direct form I transposed	• `#_of_zeros   1` • `#_of_poles   1`
Direct form II transposed	`max(#_of_zeros, #_of_poles)   1`
Biquadratic direct form II transposed (second-order sections)	`2 * #_of_filter_sections`
Lattice AR and Lattice MA	`#_of_reflection_coeffs`

Example

You can open the following model by clicking here in the MATLAB Help browser (*not* in a Web browser). To build the model yourself, follow the step-by-step instructions in “Implementing Predesigned Filters with the Digital Filter Block” on page 4-23, which also provides a full explanation of the model.

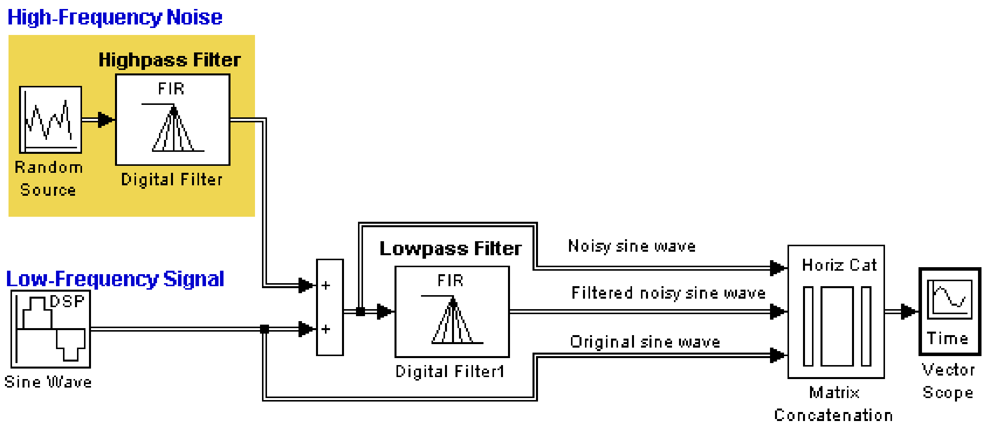


Figure 7-2: Model Using the Digital Filter Block to Implement Filters

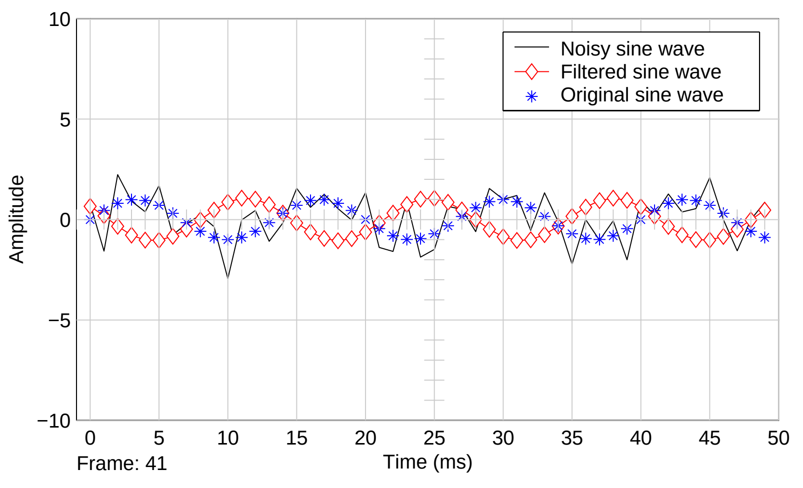
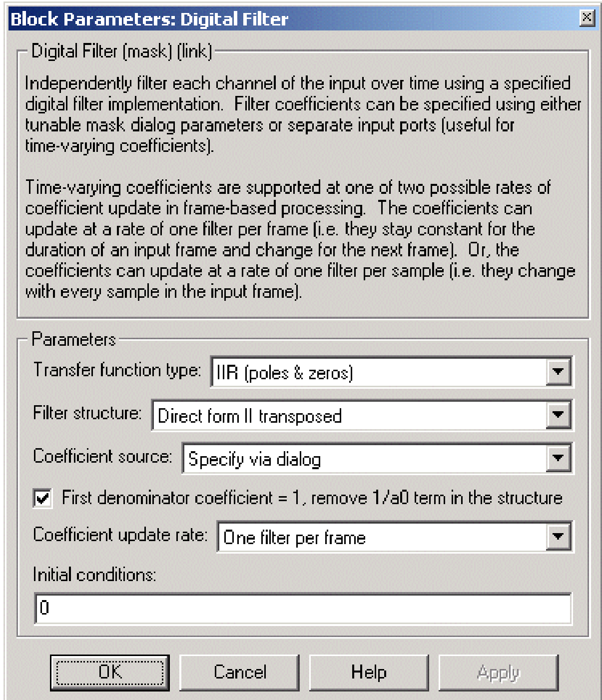


Figure 7-3: Vector Scope Display After Running the Model

Dialog Box



Transfer function type

The type of transfer function of the filter: FIR all-zero, IIR all-pole, or IIR with poles and zeros.

Filter structure

The filter's structure. The selection of structures varies depending the setting of the **Transfer function type** parameter; for the available structures, see "Supported Filter Structures" on page 7-152.

Coefficient source

Where to specify filter coefficients: in dialog parameter(s), or through input port(s). To specify *time-varying* filter coefficients that change in time, specify them through input ports (for more information on time-varying filters, see "Specifying Time-Varying Filters" on page 7-154).

Numerator coefficients

Vector of numerator coefficients of the filter's transfer function. This parameter is only visible when the **Coefficient source** parameter is set to **Dialog parameter(s)** *and* the filter lends itself to specification with numerator coefficients. Tunable.

Denominator coefficients

Vector of denominator coefficients of the filter's transfer function. This parameter is only visible when the **Coefficient source** parameter is set to **Dialog parameter(s)** *and* the filter lends itself to specification with denominator coefficients. Tunable.

Reflection coefficients

Vector of reflection coefficients of the filter's transfer function. This parameter is only visible when the **Coefficient source** parameter is set to **Dialog parameter(s)** *and* the filter lends itself to specification with reflection coefficients. Tunable.

SOS matrix (Mx6)

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 wether your SOS matrix is valid. For more on the requirements of the SOS matrix, see "Specifying the SOS Matrix (Biquadratic Filter Coefficients)" on page 7-159. This parameter is visible only when the **Coefficient source** parameter is set to **Dialog parameter(s)** *and* the filter lends itself to specification with an SOS matrix. Tunable.

First denominator coefficient = 1, remove 1/a0 term in the structure

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). 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 visible only when the **Coefficient source** parameter is set to **Input port(s)**. See "Removing the 1/a0 Term in the Filter Structure" on page 7-158 for a diagram and details.

Coefficient update rate

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. For more information, see “Specifying Time-Varying Filters” on page 7-154.

Initial conditions

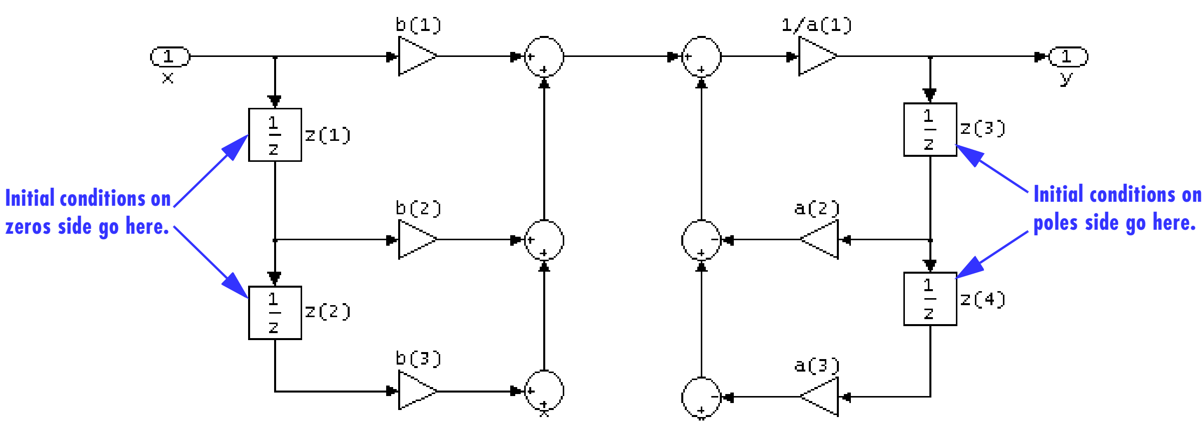
The initial conditions of the filter’s states. To learn how to specify initial conditions, see “Specifying Initial Conditions” on page 7-159.

Initial conditions on zeros side

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 following diagram. This parameter is enabled only when the filter has both poles and zeros, *and* 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 7-159.

Initial conditions on poles side

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 following diagram. This parameter is enabled only when the filter has both poles and zeros, *and* 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 7-159.



Supported Data Types

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

To learn how to convert to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

The output data type is the same as the data type of the filtered input signal (not necessarily the same as the data type of the filter coefficients).

See Also

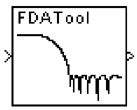
Digital Filter Design	DSP Blockset
Filter Realization Wizard	DSP Blockset
`fdatool`	Signal Processing Toolbox
`fvtool`	Signal Processing Toolbox
`sptool`	Signal Processing Toolbox

- “Filtering” on page 7-6 — List of all DSP Blockset filtering blocks
- “Implementing Predesigned Filters with the Digital Filter Block” on page 4-23 — Examples of using the Digital Filter block
- “Filters” on page 4-1 — Examples of when and how to use DSP Blockset filtering blocks

Purpose Design and implement a variety of digital FIR and IIR filters

Library Filtering / Filter Designs

Description



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 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 the 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 the Signal Processing Toolbox, and the `filter` function in Filter Design Toolbox.

Sections of This Reference Page

- "Valid Inputs and Corresponding Outputs" on page 7-168
- "Designing the Filter" on page 7-168
- "Tuning the Filter During Simulation" on page 7-168
- "Example" on page 7-169
- "Dialog Box" on page 7-170
- "Supported Data Types" on page 7-171
- "See Also" on page 7-171

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:

- "Filter Design, Analysis, and Implementation with the Digital Filter Design Block" on page 4-6
- 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.

Example

You can open the following model by clicking here in the MATLAB Help browser (*not* in a Web browser). To build the model yourself, follow the step-by-step instructions in "Example: Using the Digital Filter Design Block to Design, Analyze, and Implement a Filter" on page 4-8, which also provides a full explanation of the model.

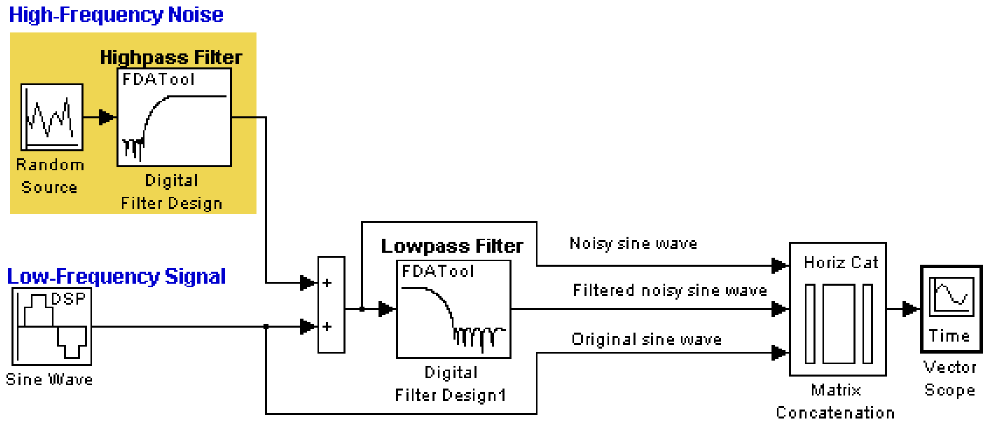


Figure 7-4: Model Using the Digital Filter Block to Implement Filters

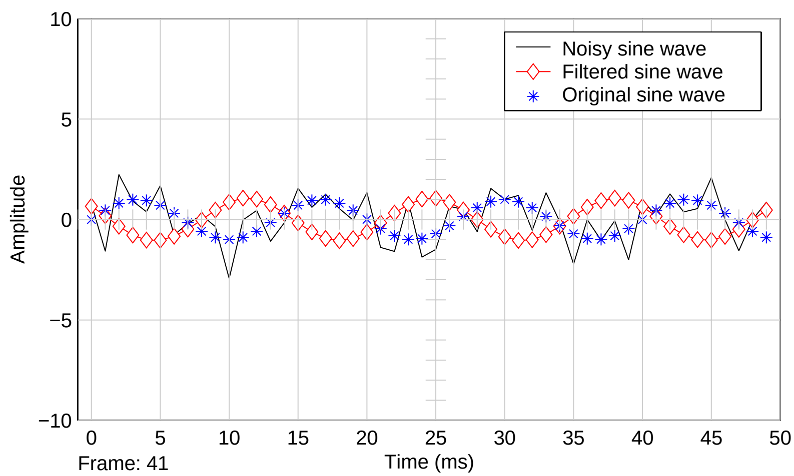
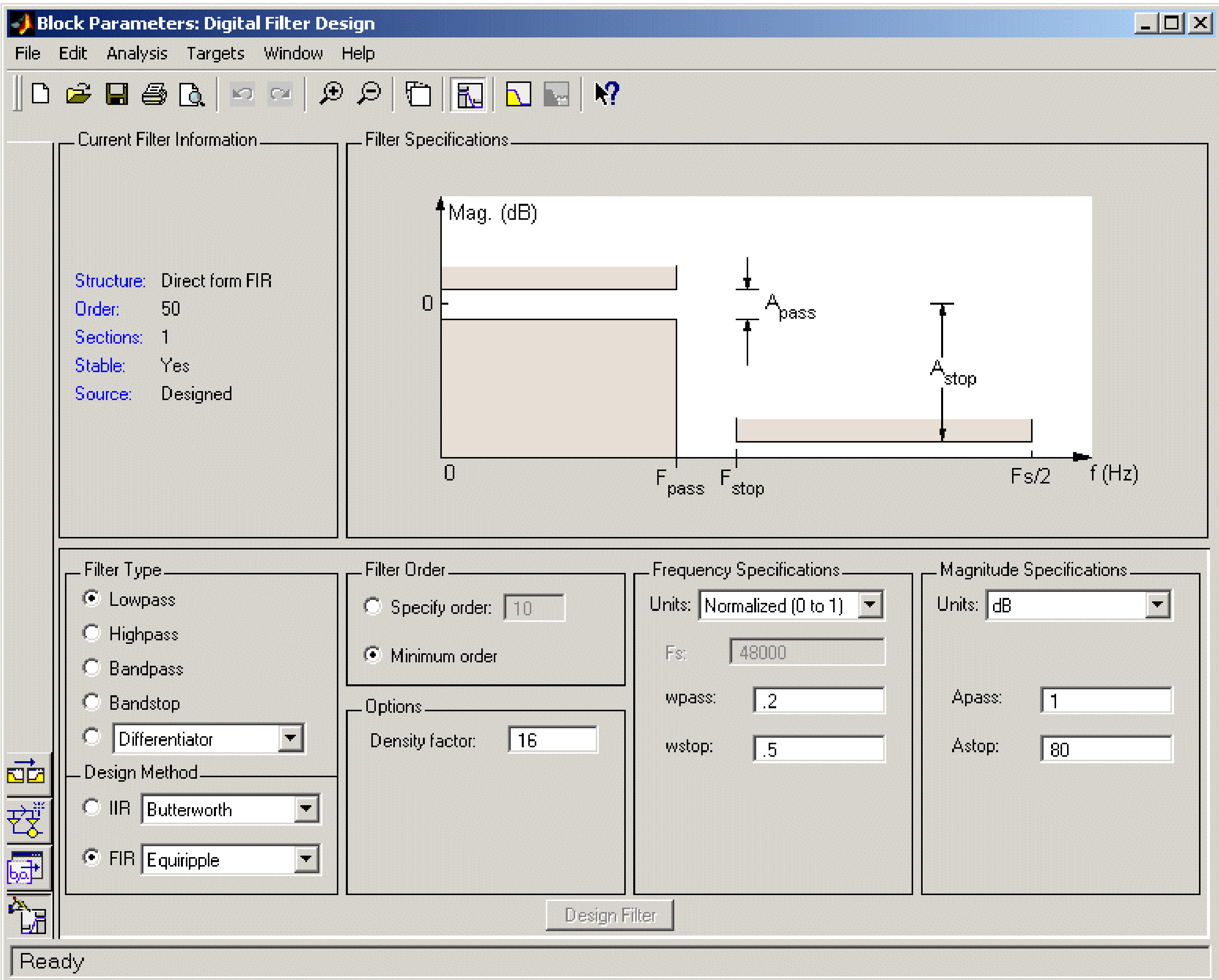
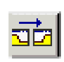


Figure 7-5: Vector Scope Display After Running the Model

Dialog Box

The FDATool GUI Opened from the Digital Filter Design Block



To get the **Transform Filter** button , install the Filter Design Toolbox. To get the **Targets** menu, install the Embedded Target for Texas Instruments C6000 DSPs.

To learn how to use the FDATool GUI, see “Designing the Filter” on page 7-168.

Supported Data Types

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

To learn how to convert to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

Analog Filter Design	DSP Blockset
Window Function	DSP 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 topics:

- "Filtering" on page 7-6 — List of all DSP Blockset filtering blocks
- "Filters" on page 4-1 — Examples of when and how to use DSP Blockset filtering blocks
- "Filter Design, Analysis, and Implementation with the Digital Filter Design Block" on page 4-6
- Topic on the Filter Design and Analysis Tool (FDATool) in the Signal Processing Toolbox documentation.

Discrete Impulse

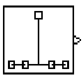
Purpose

Generate a discrete impulse.

Library

DSP Sources

Description



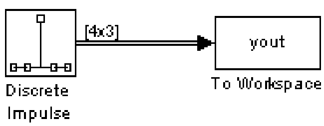
The Discrete Impulse block generates an impulse (the value `1`) at output sample `D+1`, where `D` is specified by the **Delay** parameter (`D` ≥ `0`). All output samples preceding and following sample `D+1` are zero.

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 `i`th 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 $M*T_s$.

Example

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`

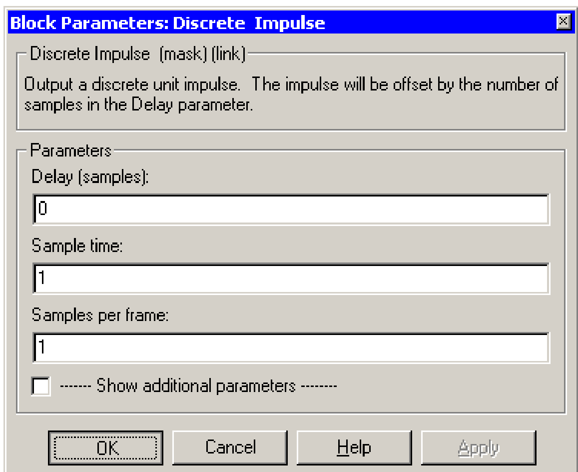
Run the model and look at the output, `yout`. The first few samples of each channel are shown below.

```
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



Delay

The number of zero-valued output samples, `D`, preceding the impulse. A length-N vector specifies an N-channel output. Tunable.

Sample time

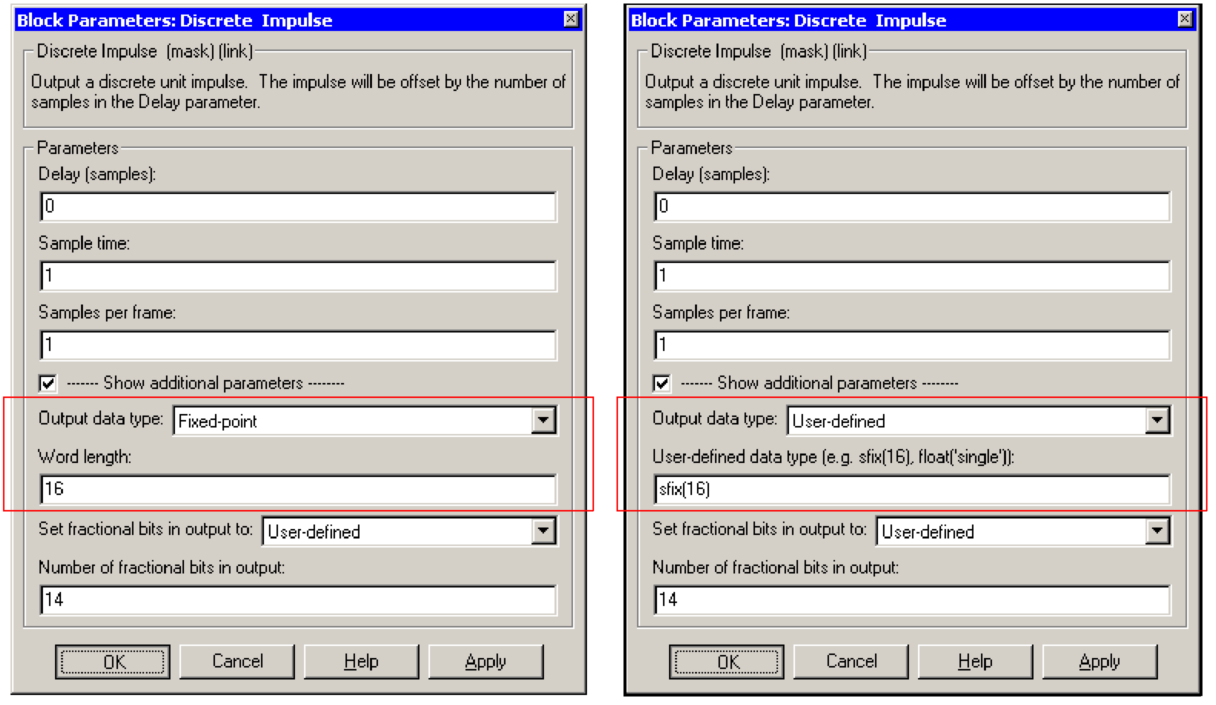
The sample period, T_s, of the output signal. The output frame period is $M*T_s$. Tunable.

Samples per frame

The number of samples, M, in each output frame. Tunable.

Show additional parameters

If selected, additional parameters specific to implementation of the block become visible as shown.



Output data type

Specify the output data type in one of the following ways:

- Choose one of the built-in data types from the drop-down list.
- Choose **Fixed-point** to specify the output data type and scaling in the **Word length**, **Set fractional bits in output to,** and **Number of fractional bits in output** parameters.
- Choose **User-defined** to specify the output data type and scaling in the **User-defined data type**, **Set fractional bits in output to,** and **Number of fractional bits in output** 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 if **Fixed-point** is selected 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 the Fixed-Point Blockset. This parameter is only visible if **User-defined** is selected for the **Output data type** parameter.

Set fractional bits 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 **Number of fractional bits in output** parameter.

This parameter is only visible if **Fixed-point** or **User-defined** is selected for the **Output data type** parameter, and if the specified output data type is a fixed-point data type.

Number of fractional bits in output

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 if **Fixed-point** or **User-defined** is selected for the **Output data type** parameter, and if **User-defined** is selected for the **Set fractional bits in output to** parameter.

Supported Data Types

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

To learn how to convert to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

Data Type Conversion	Simulink
DSP Constant	DSP Blockset
Multiphase Clock	DSP Blockset
N-Sample Enable	DSP Blockset
Signal From Workspace	DSP Blockset
`impz`	Signal Processing Toolbox

Also see the following topics:

- “Creating Signals Using Signal Generator Blocks” on page 3-36 — How to use this and other blocks to generate signals
- “DSP Sources” on page 7-3 — List of all blocks in the DSP Sources library

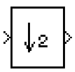
Purpose

Resample an input at a 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 $0 \le D < (K\text{-}1)$, 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 7-179 for more on the **Initial condition** parameter.

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.

The **Sample-based mode** parameter determines how the block represents the new rate at the output. There are two available options:

- **Allow multirate**

  When **Allow multirate** is selected, 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.

- **Enforce single rate**

  When **Enforce single rate** is selected, 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 **Enforce single rate** is selected 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** popup menu does not affect sample-based inputs.

Frame-Based Inputs

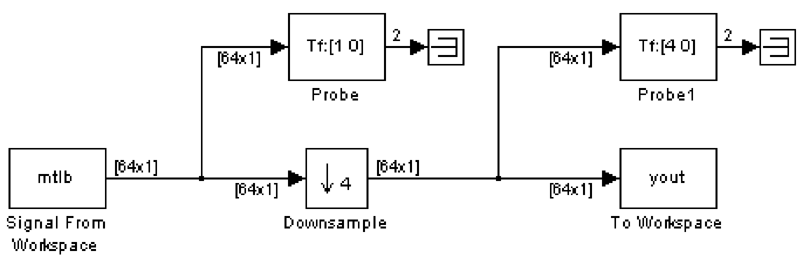
When the input is frame-based, the block treats each of the N input columns as a frame containing M_i sequential time samples from an independent channel. The block downsamples each channel independently by discarding K-1 rows of the input matrix following each row that it passes through to the output.

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.

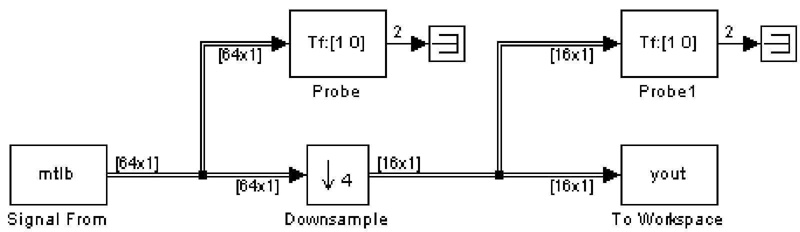
  The model below shows a single-channel input with a frame period of 1 second being downsampled by a factor of 4 to a frame period of 4 seconds. The input and output frame sizes are identical.



- **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** popup menu does not affect frame-based inputs.

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 **Enforce single rate** is selected with D > 0. The latency in this case is one sample.

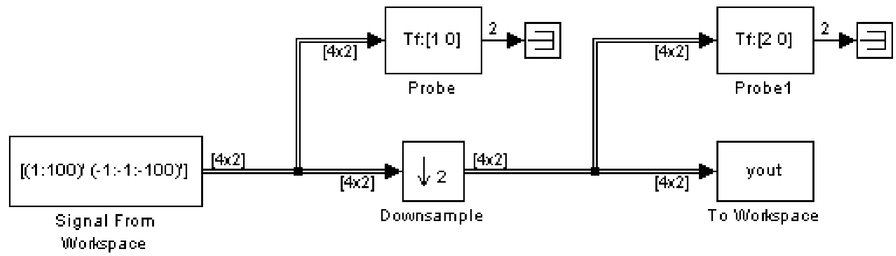
In all cases of *one-sample latency*, the initial condition for each channel appears as the first output sample. Input sample D+1 appears as the second output sample for each channel, followed by input sample D+1+K, input sample D+1+2K, and so on. The **Initial condition** parameter can be an M_i-by-N 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 example below for an illustration of this case.

See "Excess Algorithmic Delay (Tasking Latency)" on page 3-91 and "The Simulation Parameters Dialog Box" in the Simulink documentation for more information about block rates and the Simulink tasking modes.

Example

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). 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).

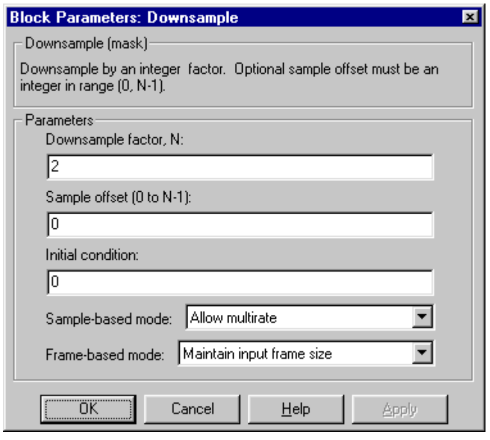
This model is multirate because there are at least two distinct frame rates, as shown by the two Probe blocks. To run this model in the Simulink multitasking mode, select **Fixed-step** and **discrete** from the **Type** controls in the **Solver** panel of the **Simulation Parameters** dialog box, and select **MultiTasking** from the **Mode** parameter. Additionally, 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
     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 (i.e., row D+1, where D is the **Sample offset**) appears in the output as sample 5 (i.e., sample M_i+1, where M_i is the input frame size).

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** (i.e, decrease the output sample rate), or **Force single-rate** (i.e., force the output sample rate to match the input sample rate by repeating every Kth 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** (i.e., decrease the frame rate), or **Maintain input frame rate** (i.e., decrease the frame size).

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 to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

FIR Decimation	DSP Blockset
FIR Rate Conversion	DSP Blockset
Repeat	DSP Blockset
Sample and Hold	DSP Blockset
Upsample	DSP Blockset

Also see "Signal Operations" on page 7-16 for a list of all the blocks in the Signal Operations library.

DSP Constant

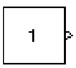
Purpose

Generate a discrete-time or continuous-time constant signal.

Library

DSP Sources

Description

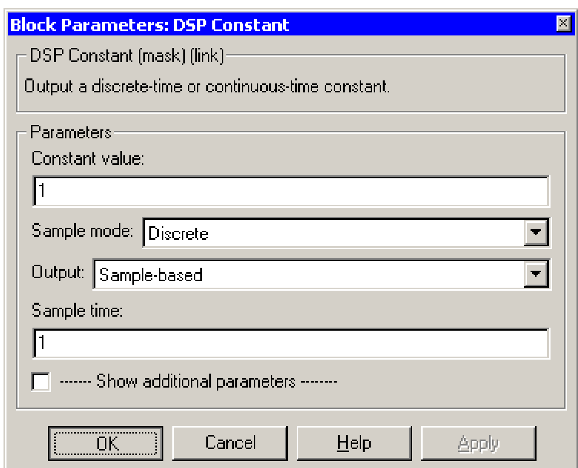


The DSP Constant block generates a signal whose value remains constant throughout the simulation. The **Constant value** parameter specifies the constant to output, and can be any valid MATLAB expression that evaluates to a scalar, vector, or matrix.

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 Box



Constant value

Specify the constant to generate. Tunable; values entered here can be tuned, but their dimensions must remain fixed.

If you specify any data type information in this field, it is overridden by the value of the **Output data type** parameter.

Sample mode

Specify the sample mode of the output, **Discrete** for a discrete-time signal or **Continuous** for a continuous-time signal.

Output

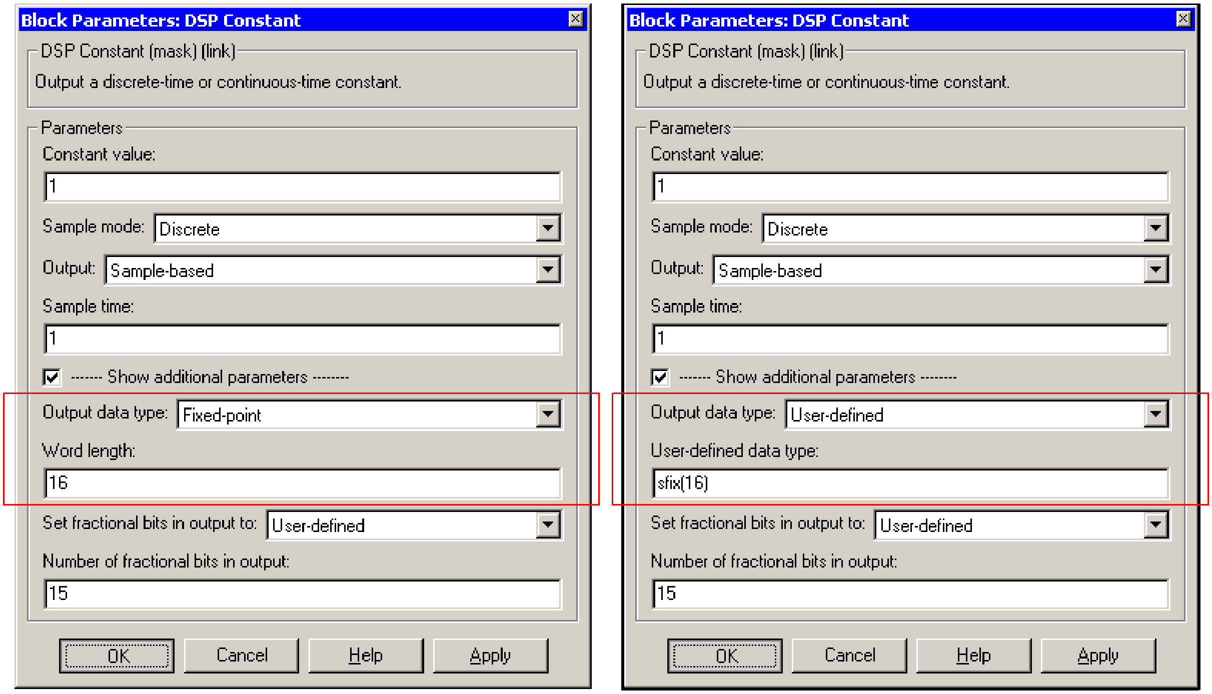
Specify whether the output is **Sample-based (interpret vectors as 1-D)**, **Sample-based**, or **Frame-based**. When **Sample-based** is selected and the output is a vector, its dimension is constrained to match the **Constant value** dimension (row or column). If **Sample-based (interpret vectors as 1-D)** is selected, however, the output has no specified dimensionality.

Sample time

Specify the discrete sample period for sample-based outputs. When **Frame-based** is selected 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 **Discrete** is selected for the **Sample mode** parameter.

Show additional parameters

If selected, additional parameters specific to implementation of the block become visible as shown.



Output data type

Specify the output data type in one of the following ways:

- Choose one of the built-in data types from the drop-down list.
- Choose **Fixed-point** to specify the output data type and scaling in the **Word length**, **Set fractional bits in output to,** and **Number of fractional bits in output** parameters.
- Choose **User-defined** to specify the output data type and scaling in the **User-defined data type**, **Set fractional bits in output to,** and **Number of fractional bits in output** parameters.
- Choose **Inherit from 'Constant value'** to set the output data type and scaling to match the values of the **Constant value** parameter.
- 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 value** parameter.

Word length

Specify the word length, in bits, of the fixed-point output data type. This parameter is only visible if **Fixed-point** is selected 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 the Fixed-Point Blockset. This parameter is only visible if **User-defined** is selected for the **Output data type** parameter.

Set fractional bits 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 **Number of fractional bits in output** parameter.

This parameter is only visible if **Fixed-point** or **User-defined** is selected for the **Output data type** parameter, and if the specified output data type is a fixed-point data type.

Number of fractional bits in output

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 if **Fixed-point** or **User-defined** is selected for the **Output data type** parameter, and if **User-defined** is selected for the **Set fractional bits in output to** parameter.

Supported Data Types

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

To learn how to convert to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

Constant	Simulink
Signal From Workspace	DSP Blockset

Also see the following topics:

- "Creating Signals Using Constant Blocks" on page 3-33 — How to use this and other blocks to generate constant signals
- "DSP Sources" on page 7-3 — List of all blocks in the DSP Sources library

Purpose
Compute the discrete wavelet transform (DWT) of the input signal

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 reference 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 detailed information about how to use this block, see the Dyadic Analysis Filter Bank block reference.

Examples
See "Examples" on page 7-199 in the Dyadic Analysis Filter Bank reference page.

See Also

Dyadic Analysis Filter Bank	DSP Blockset

Also see "Transforms" on page 7-19 for a list of all the blocks in the Transforms library.

Dyadic Analysis Filter Bank

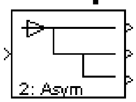
Purpose

Decompose a signal 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.)

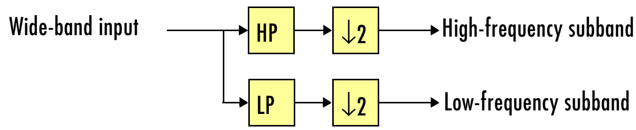
The Dyadic Analysis Filter Bank block decomposes a broadband signal into a collection of subbands with smaller bandwidths and slower sample rates. The block uses a series of highpass and lowpass FIR filters to repeatedly divide the input frequency range, as illustrated in Figure 7-6, n-Level Asymmetric Dyadic Analysis Filter Bank, on page 7-191.

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. For more information about filter banks and the block, see the other sections of this reference page.

Sections of This Reference Page

Review of 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*.

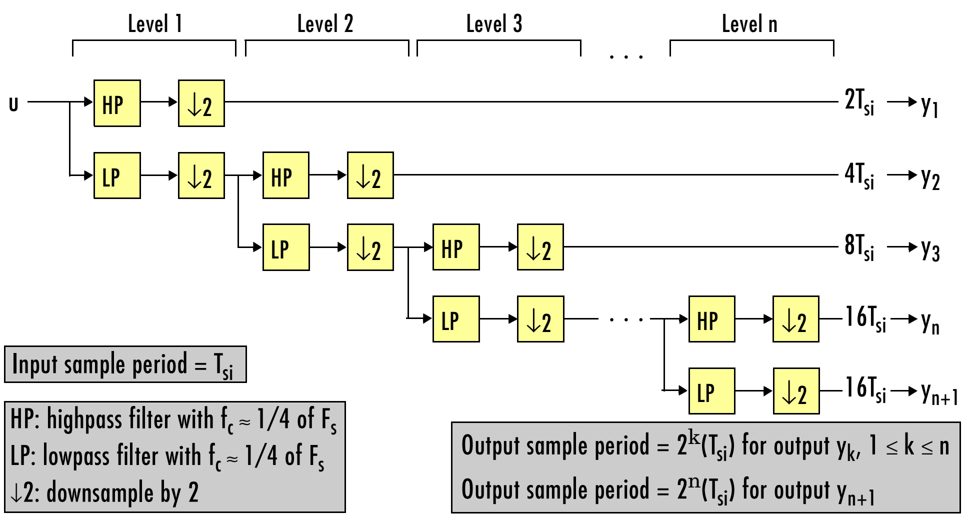


Figure 7-6: n-Level Asymmetric Dyadic Analysis Filter Bank

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.

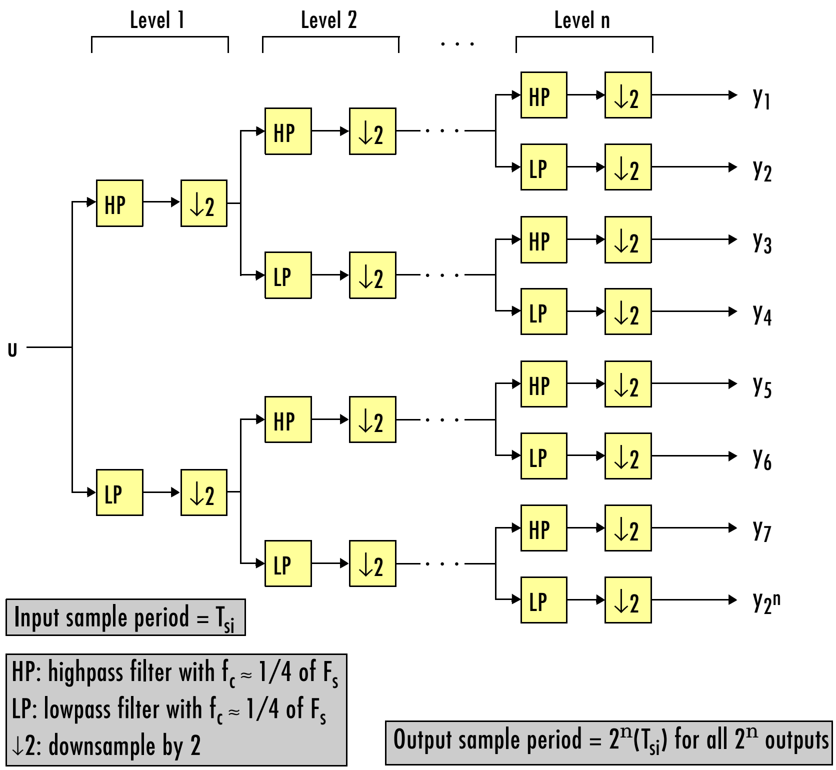


Figure 7-7: n-Level Symmetric Dyadic Analysis Filter Bank

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

Table 7-8: Notable Characteristics of Asymmetric and Symmetric Dyadic Analysis Filter Banks

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

Table 7-8: Notable Characteristics of Asymmetric and Symmetric Dyadic Analysis Filter Banks

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

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 Figure 7-8, Outputs of a 3-Level Asymmetric Dyadic Analysis Filter Bank, on page 7-195.

Output Characteristics (Setting the Output Parameter)

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 Table 7-9, Output Characteristics for n-level Dyadic Analysis Filter Bank, on page 7-196.

For more information about the filter bank levels and structures, see "Review of Dyadic Analysis Filter Banks" on page 7-191.

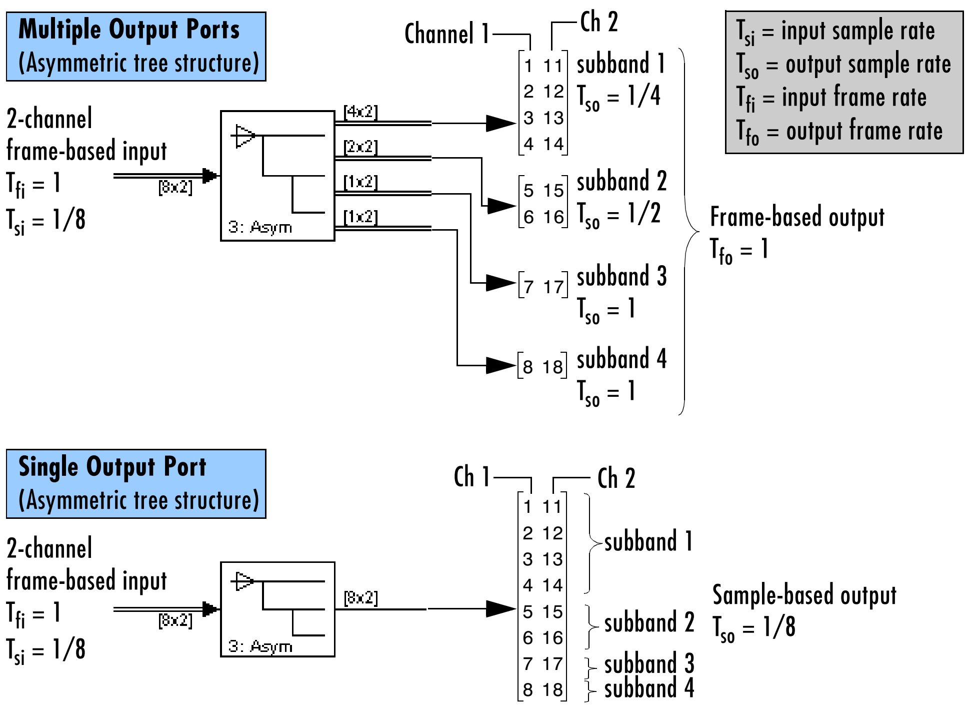


Figure 7-8: Outputs of a 3-Level Asymmetric Dyadic Analysis Filter Bank

The following table summarizes the different output characteristics of the block when it is set to output from single or multiple ports.

Table 7-9: Output Characteristics for 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 may vary, so the output sample rates may vary).

Table 7-9: Output Characteristics for n-level Dyadic Analysis Filter Bank (Continued)

	Single Output Port	Multiple Output Ports
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. $M_{o,k} = \begin{cases} M_i/2^k & (1 \le k \le n) \\ M_i/2^n & (k = n+1) \end{cases}$
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** — $F_{so,k} = \begin{cases} F_{si}/2^k & (1 \le k \le n) \\ F_{si}/2^n & (k = n+1) \end{cases}$

Specifying 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 may 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.) You must install the Wavelet Toolbox to use wavelets.

Table 7-10: 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
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')`

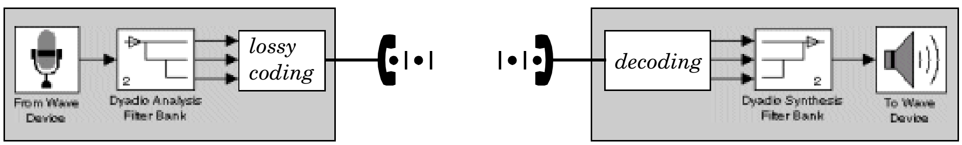
Table 7-10: Specifying Filters with the Filter Parameter and Related Parameters (Continued)

Filter	Sample Setting for Related Filter Specification Parameters	Corresponding Wavelet Function Syntax
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 DSP Blockset demos, which use the Dyadic Analysis Filter Bank block:

- Multi-level PR filter bank
- Denoising
- Wavelet transmultiplexer (WTM)

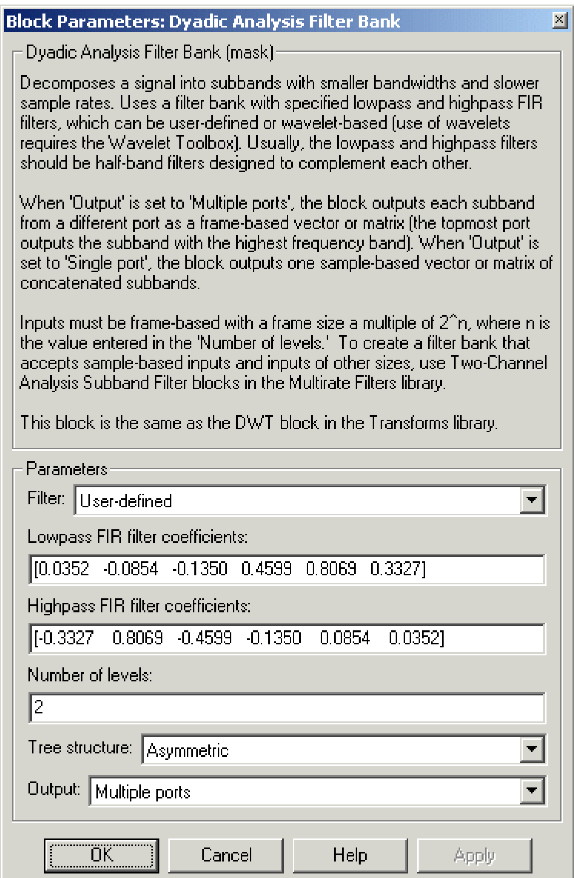
Note To see the version of the demos that use the Dyadic Analysis Filter Bank and Dyadic Synthesis Filter Bank blocks, click the **Frame-Based Demo** button in the demos.

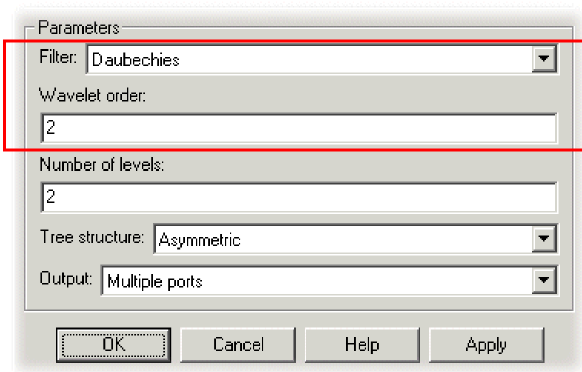
Open the demos using one of the following methods:

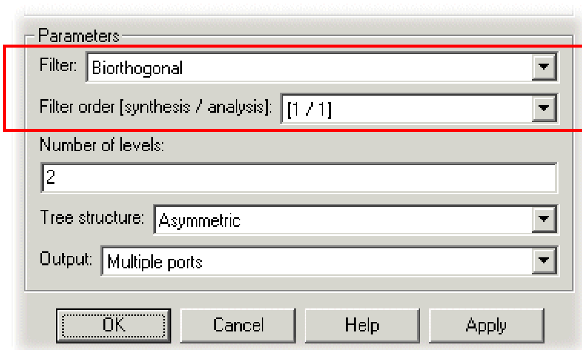
- Click the above links in the MATLAB Help browser (*not* in a Web browser).
- Type `demo blockset dsp` at the MATLAB command line, and look in the `Wavelets` directory.

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.
- 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]** may become enabled. For a list of the supported wavelets, see Table 7-10, Specifying Filters with the Filter Parameter and Related Parameters, on page 7-198.

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 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. 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 Table 7-10, Specifying Filters with the Filter Parameter and Related Parameters, on page 7-198.

Filter order [synthesis / analysis]

The order of the wavelet for the synthesis and analysis filter stages. For example, if 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 Table 7-10, Specifying Filters with the Filter Parameter and Related Parameters, on page 7-198.

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 Figure 7-6, n-Level Asymmetric Dyadic Analysis Filter Bank, on page 7-191 and Figure 7-7, n-Level Symmetric Dyadic Analysis Filter Bank, on page 7-192. 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 Figure 7-6, n-Level Asymmetric Dyadic Analysis Filter Bank, on page 7-191 and Figure 7-7, n-Level Symmetric Dyadic Analysis Filter Bank, on page 7-192.

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 (Setting the Output Parameter)" on page 7-194.

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 to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

Dyadic Synthesis Filter Bank	DSP Blockset
Two-Channel Analysis Subband Filter	DSP Blockset

See "Multirate Filters" on page 4-32 for related information. Also see "Filtering" on page 7-6 for a list of all DSP Blockset filtering blocks.

Dyadic Synthesis Filter Bank

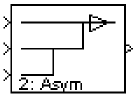
Purpose

Reconstruct a signal 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 Figure 7-9, n-Level Asymmetric Dyadic Synthesis Filter Bank, on page 7-206. 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.

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.

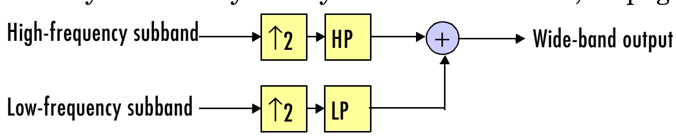
This block automatically computes wavelet-based perfect reconstruction filters if the wavelet selection in the **Filter** parameter of this block is the *same* as the **Filter** parameter setting of the corresponding Dyadic Analysis Filter Bank block. The use of wavelets requires the Wavelet Toolbox. To learn how to design your own perfect reconstruction filters, see "References" on page 7-218.

For more information about filter banks and the block, see the other sections of this reference page.

Sections of This Reference Page

Review of 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 Figure 7-9, n-Level Asymmetric Dyadic Synthesis Filter Bank, on page 7-206 and Figure 7-10, n-Level Symmetric Dyadic Synthesis Filter Bank, on page 7-207.



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*.

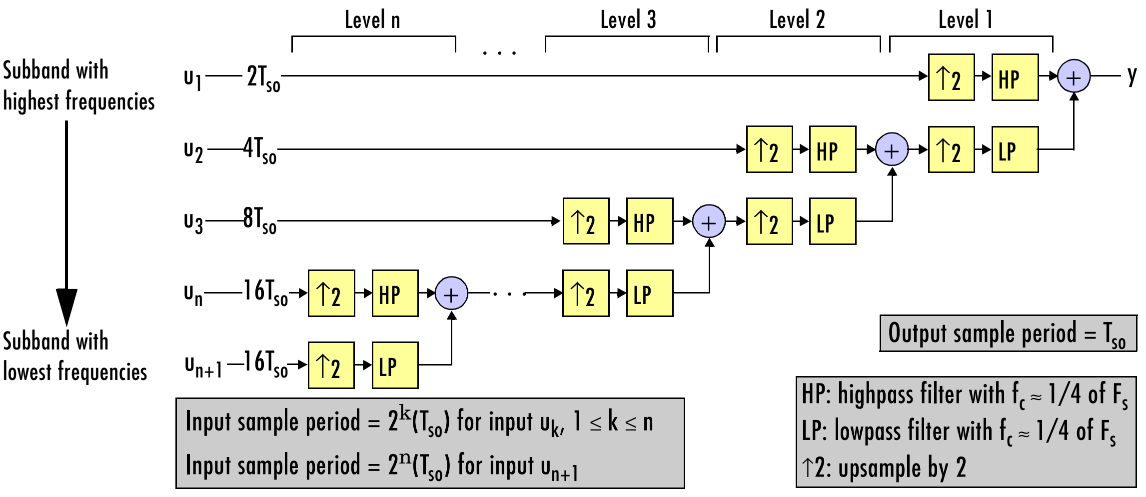


Figure 7-9: n-Level Asymmetric Dyadic Synthesis Filter Bank

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.

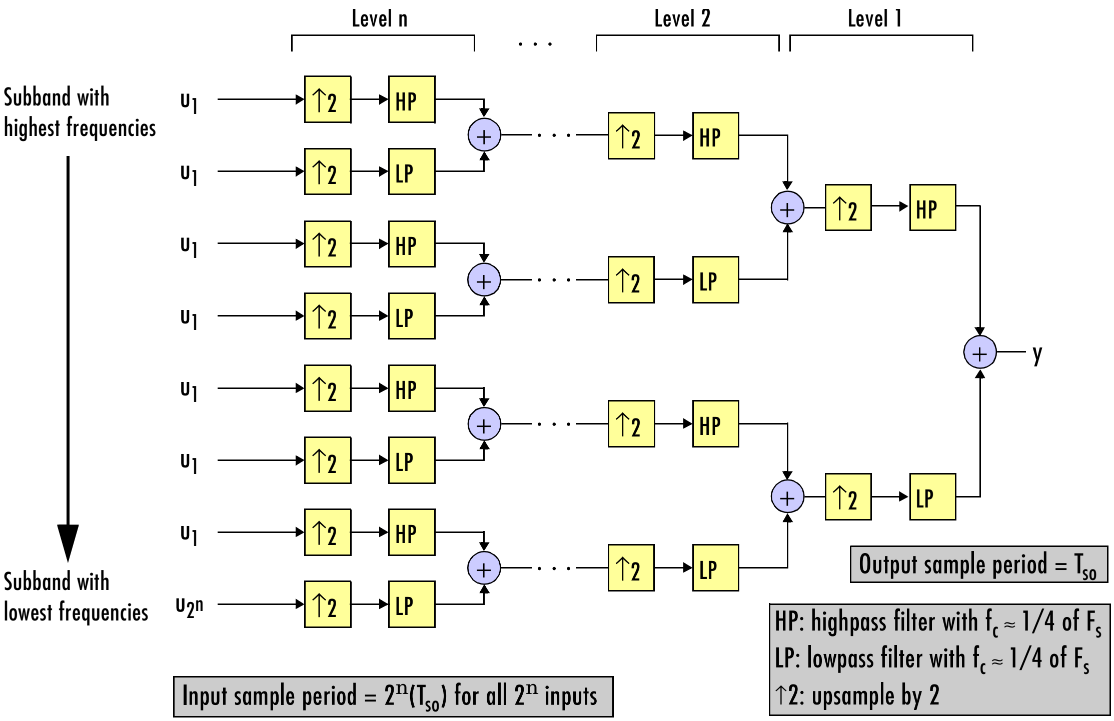


Figure 7-10: n-Level Symmetric Dyadic Synthesis Filter Bank

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

	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.
Number of Input Subbands	2^n	n+1
Bandwidth and Number of Samples in Input Subbands	All inputs subbands have bandwidth BW / 2^n and N / 2^n samples, where the output has bandwidth BW and N samples.	For an output with bandwidth BW and N samples, the *k*th input subband has the following bandwidth and number of samples. $BW_k = \begin{cases} BW/2^k & (1 \le k \le n) \\ BW/2^n & (k = n+1) \end{cases}$ $N_k = \begin{cases} N/2^k & (1 \le k \le n) \\ N/2^n & (k = n+1) \end{cases}$
Input Sample Periods	All input subbands have a sample period of $2^n(T_{so})$, where the output sample period is T_{so}.	Sample period of *k*th input subband $= \begin{cases} 2^k(T_{so}) & (1 \le k \le n) \\ 2^n(T_{so}) & (k = n+1) \end{cases}$ where the output sample period is T_{so}.

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

	n-level Symmetric	n-level Asymmetric
Total Number of Input Samples	The number of samples in the output is always equal to the total number of samples in all of the input subbands.	
Wavelet Applications	In wavelet applications, the highpass and lowpass wavelet-based filters are carefully selected so that the aliasing introduced by the decimation in the dyadic *analysis* filter bank is exactly canceled in the reconstruction of the signal in the dyadic *synthesis* filter bank.	

Input Requirements (Setting the Input Parameter)

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 = 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 = 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.

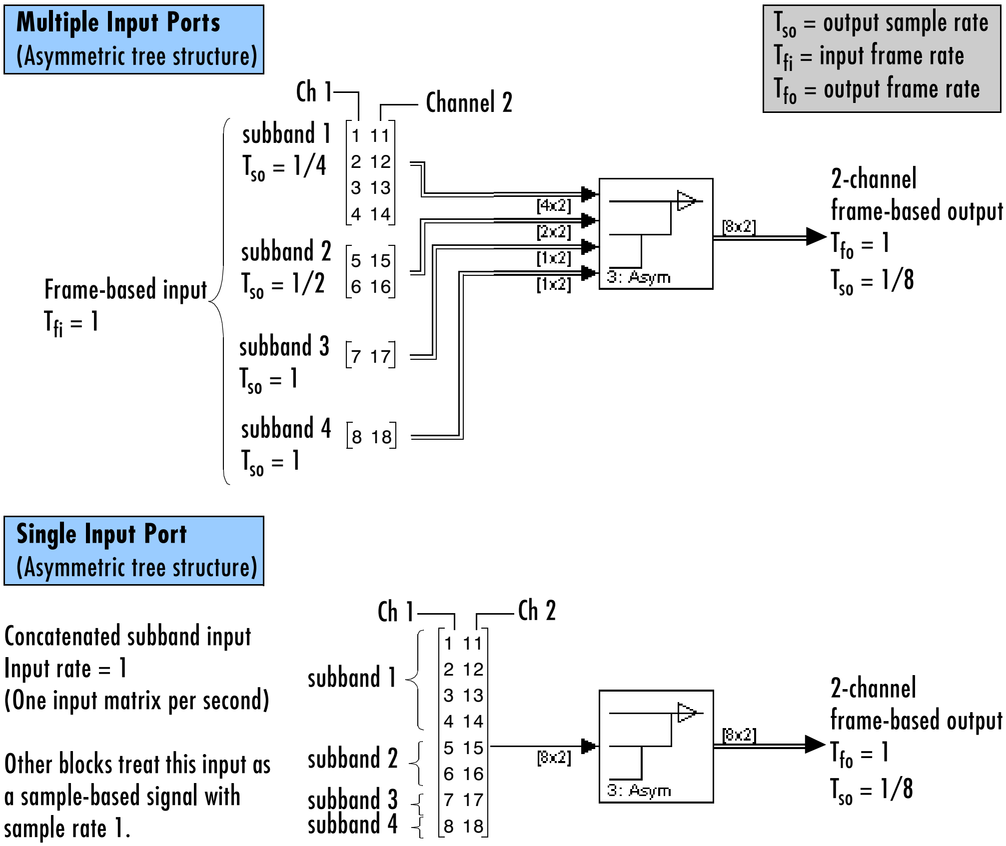


Figure 7-11: Valid Inputs to a 3-Level Asymmetric Dyadic Synthesis Filter Bank

For general information about the filter banks, see "Review of Dyadic Synthesis Filter Banks" on page 7-205.

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 Figure 7-11, Valid Inputs to a 3-Level Asymmetric Dyadic Synthesis Filter Bank, on page 7-211.

<table>
<tr><th></th><th>Frame-Based Inputs
(Input = Multiple ports)</th><th>Concatenated Subband Inputs
(Input = Single port)</th></tr>
<tr><td>Output Frame Status</td><td colspan="2">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.</td></tr>
<tr><td>Output Frame Rate</td><td>Same as the input frame rate.</td><td>Same as the input rate (the rate of the concatenated subband inputs).</td></tr>
<tr><td>Output Frame Dimensions</td><td>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.</td><td>The output has the same number of rows and columns as the input.</td></tr>
</table>

For general information about the filter banks, see “Review of Dyadic Synthesis Filter Banks” on page 7-205.

Specifying 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 7-218.
- 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 may 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.

Table 7-11: 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
Haar	None	`wfilters('haar')`
Daubechies	**Wavelet order** = `4`	`wfilters('db4')`

Table 7-11: Specifying Filters with the Filter Parameter and Related Parameters (Continued)

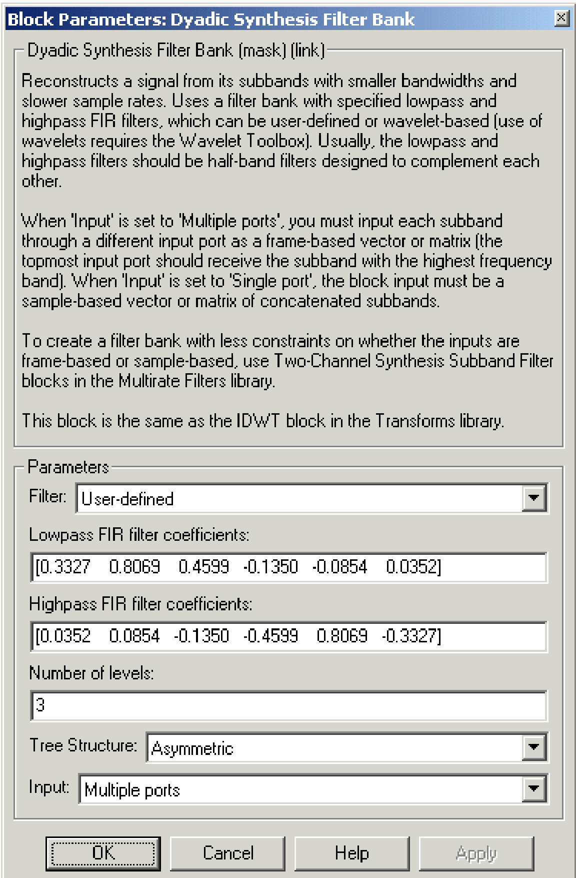
Filter	Sample Setting for Related Filter Specification Parameters	Corresponding Wavelet Function Syntax
Symlets	**Wavelet order** = `3`	`wfilters('sym3')`
Coiflets	**Wavelet order** = `1`	`wfilters('coif1')`
Biorthogonal	**Filter order [synthesis / analysis]** = `[3/1]`	`wfilters('bior3.1')`
Reverse Biorthogonal	**Filter order [synthesis / analysis]** = `[3/1]`	`wfilters('rbio3.1')`
Discrete Meyer	None	`wfilters('dmey')`

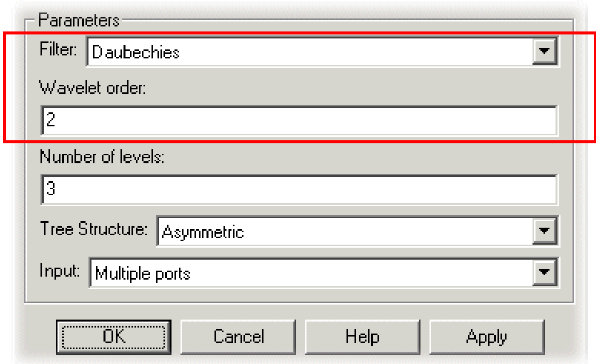
Examples

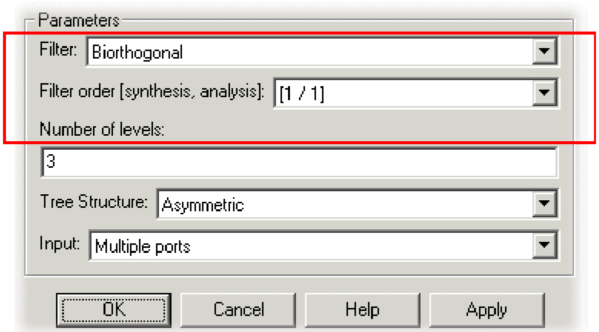
See "Examples" on page 7-199 in the Dyadic Analysis Filter Bank block reference.

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.







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]** may become enabled. For a list of the supported wavelets, see Table 7-11, Specifying Filters with the Filter Parameter and Related Parameters, on page 7-213.

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 Table 7-11, Specifying Filters with the Filter Parameter and Related Parameters, on page 7-213.

Filter order [synthesis / analysis]

The order of the wavelet for the synthesis and analysis filter stages. For example, if 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 Table 7-11, Specifying Filters with the Filter Parameter and Related Parameters, on page 7-213.

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 Figure 7-9, n-Level Asymmetric Dyadic Synthesis Filter Bank, on page 7-206 and Figure 7-10, n-Level Symmetric Dyadic Synthesis Filter Bank, on page 7-207. 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 Figure 7-9, n-Level Asymmetric Dyadic Synthesis Filter Bank, on page 7-206 and Figure 7-10, n-Level Symmetric Dyadic Synthesis Filter Bank, on page 7-207.

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 (Setting the Input Parameter)" on page 7-210.

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 to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Dyadic Analysis Filter Bank	DSP Blockset
Two-Channel Synthesis Subband Filter	DSP Blockset

See “Multirate Filters” on page 4-32 for related information. Also see “Filtering” on page 7-6 for a list of all DSP Blockset filtering blocks.

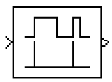
Purpose

Detect a transition of the input from zero to a nonzero value.

Library

Signal Management / Switches and Counters

Description

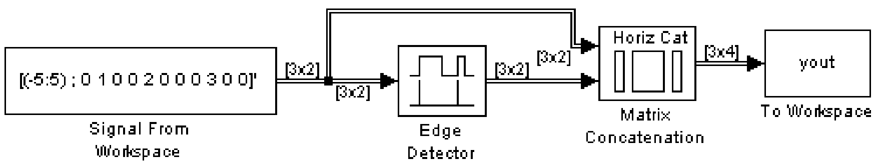


The Edge Detector block generates an impulse (the value 1) in a given output channel when the corresponding channel of the input transitions from zero to a nonzero value. Otherwise, the block generates zeros in each channel.

The output has the same dimension and sample rate as the input. If 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 (i.e., 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.

Example

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 `yout`.

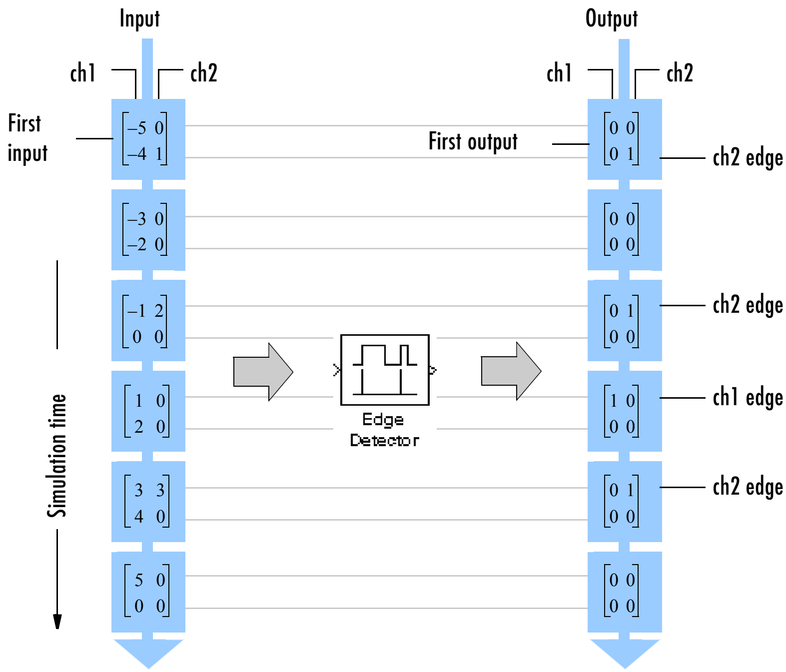


Adjust the block parameters as described below. (Use the default settings for the To Workspace block.)

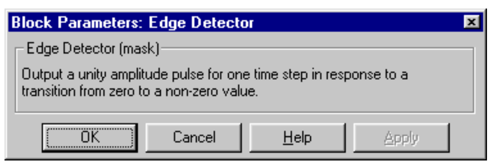
- Set the Signal From Workspace block parameters as follows:
  - **Signal** = `[(-5:5) ; 0 1 0 0 2 0 0 0 3 0 0]'`
  - **Sample time** = `1`
  - **Samples per frame** = `3`
- Set the Matrix Concatenation block parameters as follows:
  - **Number of inputs** = `2`
  - **Concatenation method** = **Horizontal**

As shown below, the block finds edges at sample 7 in channel 1, and at samples 2, 5, and 9 in channel 2.

Edge Detector



Dialog Box



Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed-point
- Custom data types
- Boolean — The block may 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 A-11. To learn how to disable Boolean output support, see "Steps to Disabling Boolean Support" on page A-12.
- 8-, 16-, and 32-bit signed integers

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

To learn how to convert to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

Counter	DSP Blockset
Event-Count Comparator	DSP Blockset

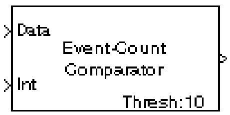
Also see "Switches and Counters" on page 7-15 for a list of all the blocks in the Switches and Counters library.

Event-Count Comparator

Purpose Detect threshold crossing of accumulated nonzero inputs.

Library Signal Management / Switches and Counters

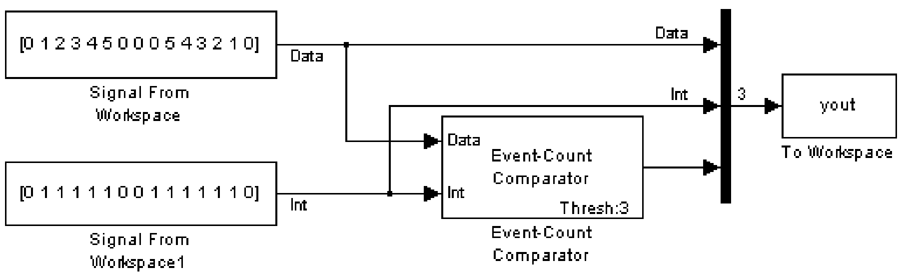
Description The Event-Count Comparator block records the number of nonzero inputs to the `Data` port during the period that the block is enabled by a high signal (the value `1`) at the interval (`Int`) port. Both inputs must be scalars, and the `Int` input must be sample-based.



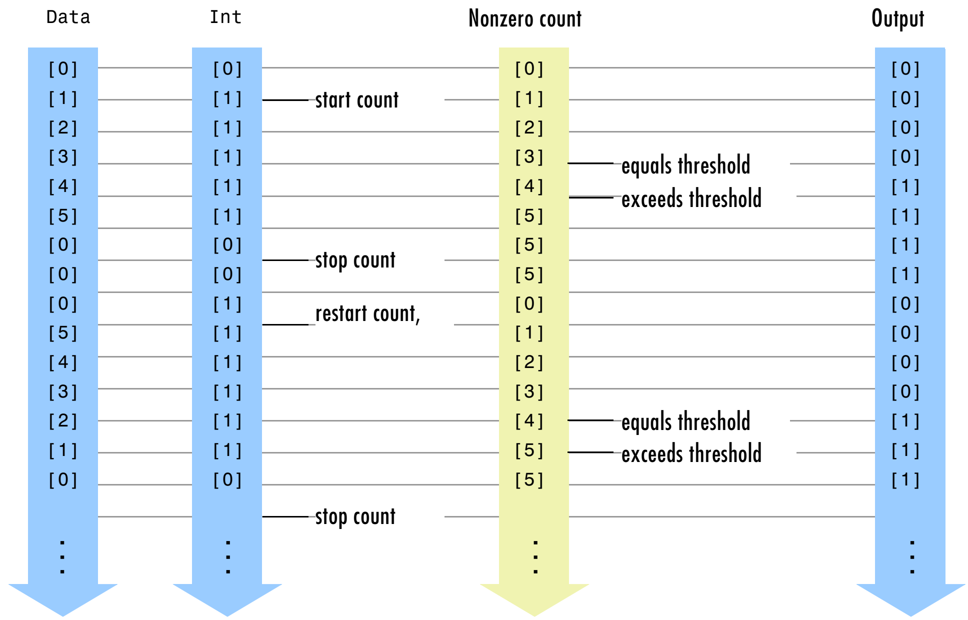
When the number of accumulated nonzero inputs first equals the **Event threshold** setting, the block waits one additional sample interval, and then sets the output high (`1`). The block holds the output high until recording is restarted by a low-to-high (`0`-to-`1`) transition at the `Int` port.

If the input to the `Data` port is frame-based, the output is frame-based; otherwise, the output is sample-based.

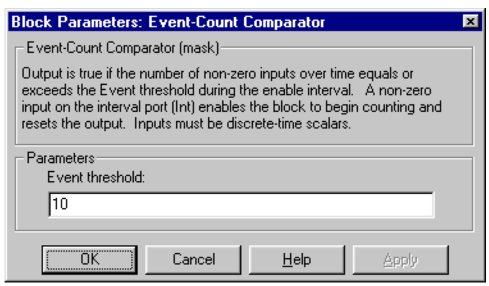
Example In the model below, the Event-Count Comparator block (**Event threshold** = 3) detects two threshold crossings in the input to the `Data` port, one at sample 4 and one at sample 12.



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

The value against which to compare the number of nonzero inputs. Tunable.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Boolean — The block may 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 A-11. To

learn how to disable Boolean output support, see “Steps to Disabling Boolean Support” on page A-12.

To learn how to convert to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Counter	DSP Blockset
Edge Detector	DSP Blockset

Also see “Switches and Counters” on page 7-15 for a list of all the blocks in the Switches and Counters library.

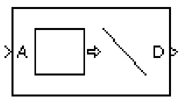
Purpose

Extract the main diagonal of the input matrix.

Library

Math Functions / Matrices and Linear Algebra / Matrix Operations

Description

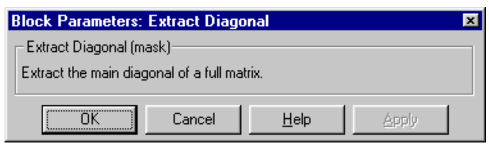


The Extract Diagonal block populates the 1-D output vector with the elements on the main diagonal of the M-by-N input matrix `A`.

```
D = diag(A)        Equivalent MATLAB code
```

The output vector has length `min(M,N)`, and is always sample-based.

Dialog Box



Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed-point
- Custom data types
- Boolean — Block outputs are always Boolean. To learn how to disable Boolean support, see “Steps to Disabling Boolean Support” on page A-12.
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

To learn how to convert to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Constant Diagonal Matrix	DSP Blockset
Create Diagonal Matrix	DSP Blockset
Extract Triangular Matrix	DSP Blockset
`diag`	MATLAB

Also see “Matrix Operations” on page 7-11 for a list of all the blocks in the Matrix Operations library.

Extract Triangular Matrix

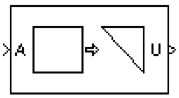
Purpose

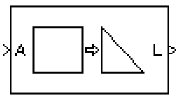
Extract the lower or upper triangle from an input matrix.

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.

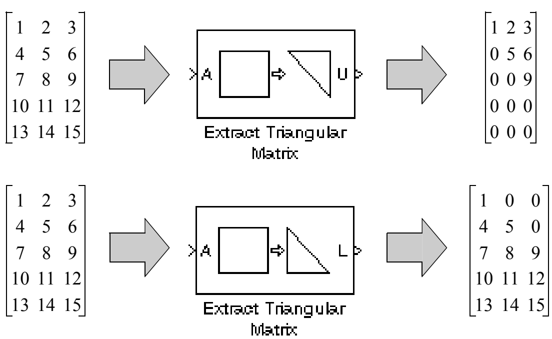
The **Extract** parameter selects between the two components of the input:

- **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.

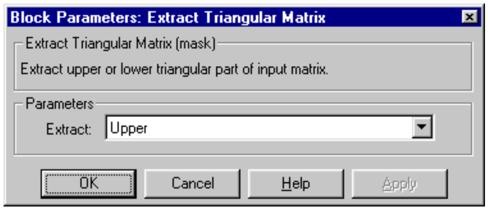
The output has the same frame status as the input.

Example

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, except in the Simulink external mode.

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 to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

Autocorrelation LPC	DSP Blockset
Cholesky Factorization	DSP Blockset
Constant Diagonal Matrix	DSP Blockset
Extract Diagonal	DSP Blockset
Forward Substitution	DSP Blockset
LDL Factorization	DSP Blockset
LU Factorization	DSP Blockset
`tril`	MATLAB
`triu`	MATLAB

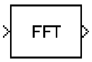
Also see "Matrix Operations" on page 7-11 for a list of all the blocks in the Matrix Operations library.

FFT

Purpose Compute the FFT of the input.

Library Transforms

Description



The FFT block computes the fast Fourier transform (FFT) of each channel in the 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 is always complex-valued and sample-based (it is an unoriented 1-D vector for unoriented inputs).

```
y = fft(u,M)        % Equivalent MATLAB code
```

The kth entry of the lth output channel, $y(k, l)$, is the kth point of the M-point discrete Fourier transform (DFT) of the lth input channel.

$$y(k,l) = \sum_{m=1}^{M} u(m,l)e^{-j2\pi(m-1)(k-1)/M} \qquad k = 1, \ldots, M \tag{7-2}$$

For information on block output characteristics and how to configure the block computation methods, see other sections of this reference page.

Sections of This Reference Page

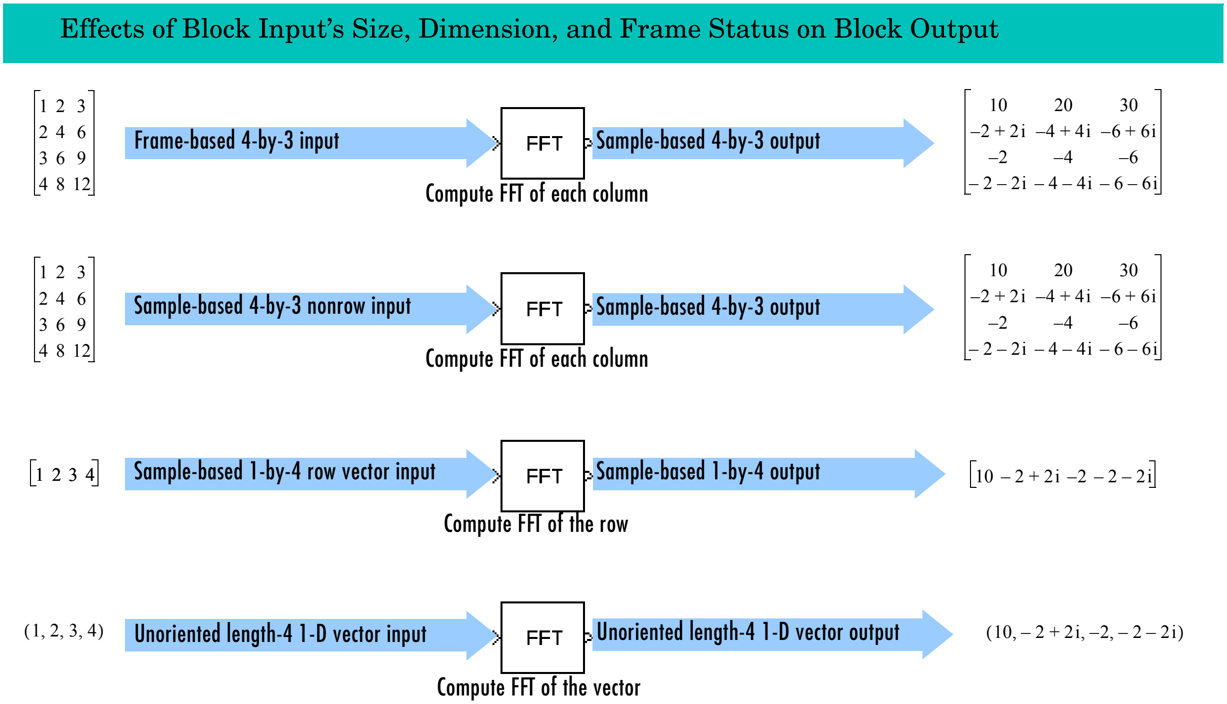
Input and Output Characteristics

The following table describes all valid block input types, their corresponding outputs, and the dimension along which the block computes the DFT.

Note For M-by-N and length-M inputs, 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. Also, to get valid outputs, your inputs must be in linear order.

<table>
<tr><th>Valid Block Inputs
• Real- or complex-valued
• Must be in linear order
• M must be a power of two</th><th>Dimension Along Which Block Computes DFT</th><th>Corresponding Block Output Characteristics
Output port rate = input port rate</th></tr>
<tr><td>Frame-based M-by-N matrix</td><td>Column</td><td rowspan="3">• Sample-based
• Complex-valued
• Same dimensions as input
• Each column (each row for sample-based row inputs) contains the M-point DFT of the corresponding input channel in linear or bit-reversed order.</td></tr>
<tr><td>Sample-based M-by-N matrix, $M \neq 1$</td><td>Column</td></tr>
<tr><td>Sample-based 1-by-M row vector</td><td>Row</td></tr>
<tr><td>Unoriented length-M 1-D vector</td><td>Vector</td><td>Unoriented, length-M, complex-valued 1-D vector containing M-point DFT of input in linear or bit-reversed order</td></tr>
</table>

Click here in the MATLAB Help Browser to open a Simulink model based on the following diagram.



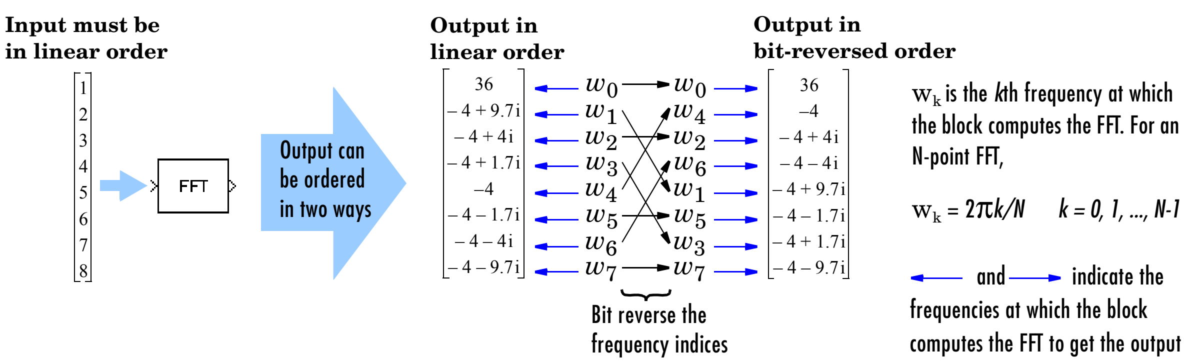
Ordering Output Column Entries (Output in bit-reversed order Parameter)

Set the **Output in bit-reversed order** parameter as follows to indicate the ordering of the output's column elements. For a definition of bit-reversed ordering, see "Description of Bit-Reversed Ordering" on page 7-232.

Parameter Setting	Ordering of Output Channel Elements	Output Column Entries
	Linear order (See the following note.)	*k*th column element is the DFT of the corresponding input column at the *k*th frequency.
	Bit-reversed order	*k*th column element is the DFT of the corresponding input column at the *r*th frequency, where *r* is the bit reversed value of *k*.

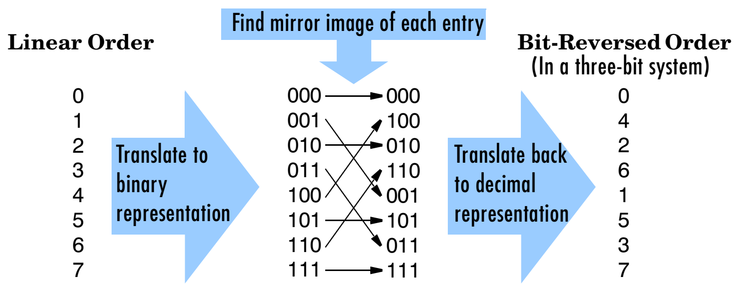
Note Linearly ordering the output requires extra data sorting manipulation, so in some situations it may be better to output in bit-reversed order as illustrated in the example, "Use of Outputs in Bit-Reversed Order" on page 7-235.

The next diagram illustrates the difference between linear and bit-reversed outputs. Note that output values in linear and bit-reversed order are the same; only the order in which they appear in the columns differs.



Description of Bit-Reversed Ordering. 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.

The sequence 0, 1, 2, 3, 4, 5, 6, 7, is in *linear order*. To put the sequence in *bit-reversed order*, replace each element in the linearly ordered sequence with its bit-reversed counterpart. You can do this by translating the sequence into its binary representation (with the minimum number of bits), then finding the mirror image of each binary entry, and translating the sequence back to its decimal representation. The resulting sequence is the original linearly ordered sequence in 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 $e^{-j2\pi(m-1)(k-1)/M}$ in Equation 7-2. This parameter has two settings, each with its advantages and disadvantages, as described in the following table.

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" below.
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 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	$3N/4$	1536 bytes
Memory	$N/4 + 1$	516 bytes

Algorithms Used for FFT Computation

Depending on whether the block input is 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 next table:

- Radix-2 decimation-in-time (DIT) algorithm
- Radix-2 decimation-in-frequency (DIF) algorithm
- Half-length algorithm
- Double-signal algorithm

Input Complexity	Output Ordering	Algorithms Used for FFT Computation
Complex	Linear or bit-reversed	Radix-2 DIT
Real	Linear	Radix-2 DIT in conjunction with the half-length and double-signal algorithms when possible
Real	Bit-reversed	Radix-2 DIF in conjunction with the half-length and double-signal algorithms when possible

Example

Use of Outputs in Bit-Reversed Order. The FFT block runs more quickly when it outputs in bit-reversed order. You can often use an output in bit-reversed order when your model also uses the IFFT block (the IFFT block allows you to indicate whether its input is in bit-reversed or linear order). For instance, set the FFT block to output in bit-reversed order when you want to filter or convolve signals by taking the FFT of time domain data, multiplying frequency-domain data, and inputting the product to an IFFT block.

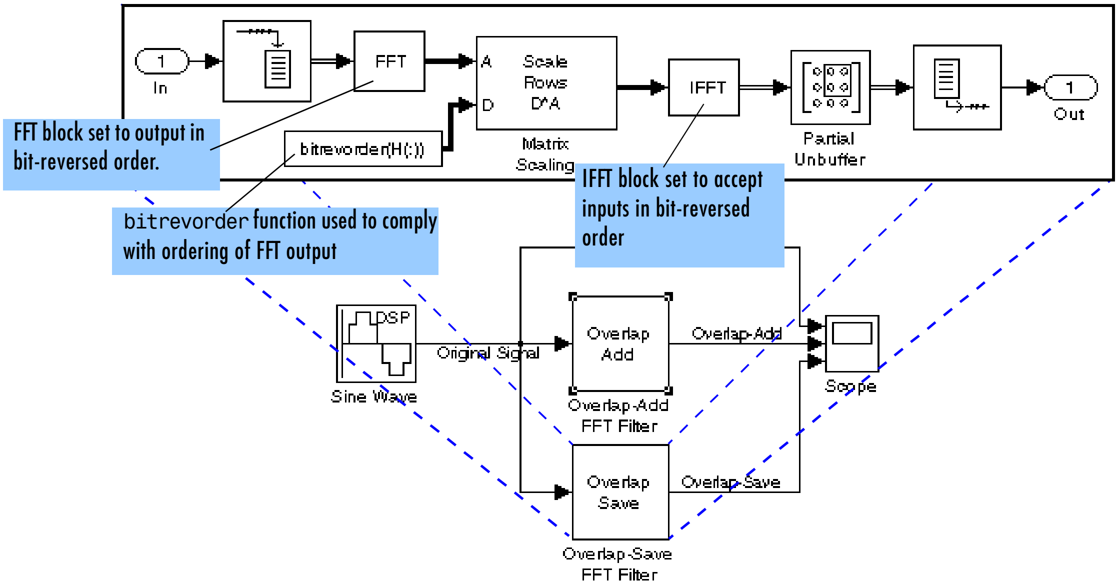
The following model shows the implementation of the Overlap-Save FFT Filter block. The implementation uses the FFT block in conjunction with an IFFT block, so the FFT block is set to output in bit-reversed order, and the IFFT block is set to accept inputs in bit-reversed order. Note the implementation uses the `bitrevorder` function to put the vector H into bit-reversed order before multiplying it with the bit-reversed FFT outputs.

1. Click here in the MATLAB Help browser to open the demo model. Alternatively, type the following command at the MATLAB command line.

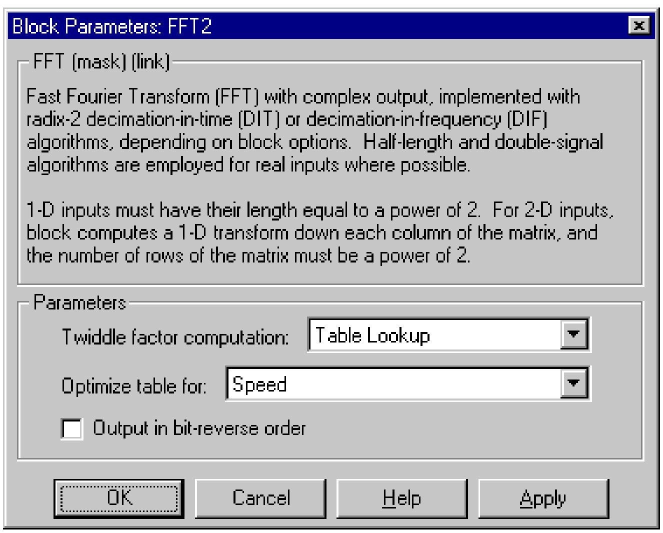
   ```
   olapfilt           % Open the following demo model
   ```

2. To see the implementation of the Overlap-Save FFT Filter block, right-click on the Overlap-Save FFT Filter block, and select **Look under mask**.
3. Look under the mask of the Overlap-Add FFT Filter block as well, which also uses an FFT block that outputs in bit-reversed order.

Implementation of the Overlap-Save FFT Filter Block



Dialog Box



Twiddle factor computation

Computation method of the term $e^{-j2\pi(m-1)(k-1)/M}$ in Equation 7-2. 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 7-233.

Optimize table for

Optimization of the table of sine and cosine values for **Speed** or **Memory**. Active only when **Twiddle factor computation** is set to **Table lookup**. See "Selecting the Twiddle Factor Computation Method" on page 7-233.

Output in bit-reverse order

Order of the output channel elements relative to the ordering of the input elements. When checked, 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. See "Ordering Output Column Entries (Output in bit-reversed order Parameter)" on page 7-231.

Supported Data Types

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

To learn how to convert to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

Complex Cepstrum	DSP Blockset
DCT	DSP Blockset
IFFT	DSP Blockset
Pad	DSP Blockset
Zero Pad	DSP Blockset
`bitrevorder`	Signal Processing Toolbox
`fft`	Signal Processing Toolbox
`ifft`	Signal Processing Toolbox

Also see "Transforms" on page 7-19 for a list of all the blocks in the Transforms library.

Filter Realization Wizard

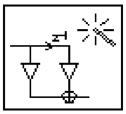
Purpose

Construct filter realizations using Sum, Gain, and Integer Delay blocks

Library

Filtering / Filter Designs

Description



Note Use this block to implement fixed-point or floating-point digital filters built from Sum, Gain, and Integer Delay blocks. 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 efficiently implement floating-point filters that you have already designed
- Digital Filter Design — Use to design, analyze, and then efficiently 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 creates a subsystem block that implements the specified filter using Sum, Gain, and Integer 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 any frame-based row vector signal, and outputs the result.

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 all of the powerful tools in FDATool to design and analyze your filter, and then use the Filter Realization Wizard parameters to implement the filter in your models.

To learn how to use the Filter Realization Wizard, see other sections of this reference page.

Sections of This Reference Page

Valid Inputs and Corresponding Outputs

The Filter Realization Wizard creates a new a subsystem block that implements the specified filter. The subsystem 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, or frame-based row vectors.

Corresponding Outputs. The output of the subsystem block has the same dimensions and frame status as the input.

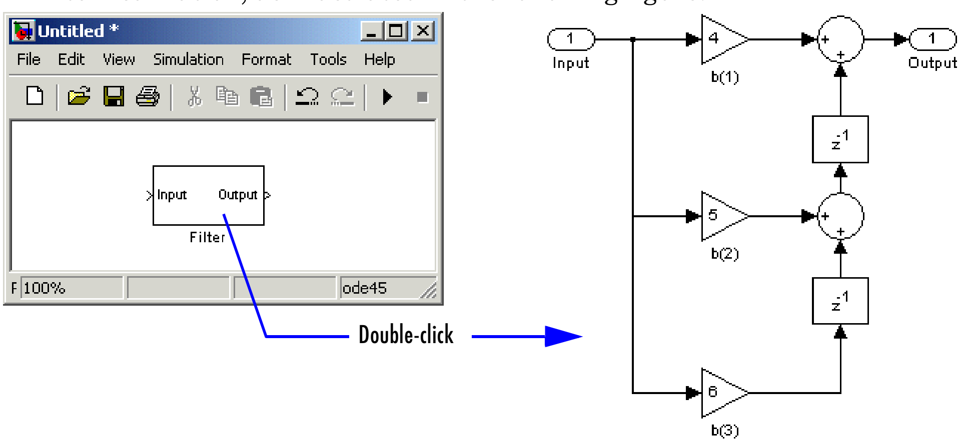
What is Considered an Independent Channel. The subsystem block treats each *element* of a vector or matrix as an independent channel.

Steps to Implementing a Filter with This Block

The parameters for this block are in a panel of the Filter Design and Analysis Tool (FDATool) GUI. In addition to using FDATool's Filter Realization Wizard parameters in the **Realize Model** panel, you must also use other panels to specify your filter. (To access the different panels, use the sidebar buttons in the lower-left corner of FDATool.)

To implement a filter using the Filter Realization Wizard, you must do the following:

1. Open the filter realization wizard by typing `dspfwiz`, or double-clicking the block in the Filter Designs library or in a model.
2. Specify a filter (including its data type support) by either designing a filter using the **Design Filter** panel, or by importing a filter using the **Import Filter** panel. To learn how, see "Specifying the Filter and Its Data Type Support" on page 7-241.
3. Optionally change the default filter structure and the default number of filter sections. To learn how, see "Setting the Filter Structure and Number of Filter Sections" on page 7-242.
4. Do the following in the **Realize Model** panel:
   - Select the destination and the name of the filter subsystem block, and whether to overwrite a filter created previously by the Filter Realization Wizard. For details, see "Setting Where to Put the Filter" on page 7-244.
   - Select the filter structure optimizations. To learn more, see "Optimizing the Filter Structure" on page 7-245.
   - Set the data type of the input, output, and computations in the filter *implementation* to match those of the specified filter. To learn how, see "Setting the Data Type of the Filter Implementation" on page 7-246.
5. Click **Realize model** in the **Realize Model** panel to realize the filter. A new block appears in a specified model. Double-click the new block to see the filter realization, as illustrated in the following figure.



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 Signal Processing Toolbox and Filter Design Toolbox.)

You can also specify a fixed-point filter, a single-precision filter, or a filter with different input, output, and computation data types. You can specify such filters by using the **Set Quantization Parameters** panel, or by importing a `qfilt` filter object with the **Import Filter** panel. Both of these options require the Filter Design Toolbox.

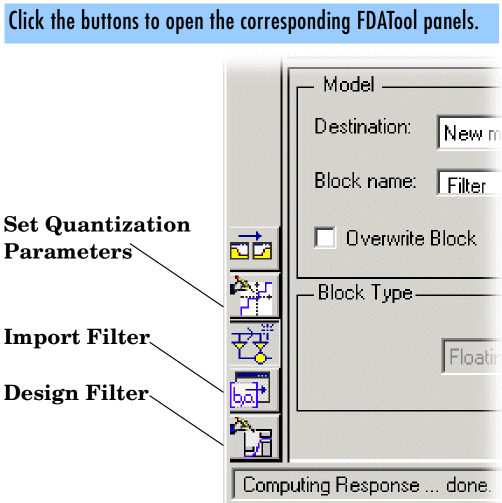
To learn how to ensure that your filter *implementation* reflects the data types of the filter you specify, see "Setting the Data Type of the Filter Implementation" on page 7-246.

Note *Running* a model containing implementations of non-double-precision filters requires the Fixed-Point Blockset, but you can still edit models containing such filter implementations without the Fixed-Point Blockset. For more information, see the topic on licensing information in the Fixed-Point Blockset documentation.

See the following topics to learn how to use the panels to specify your filter:

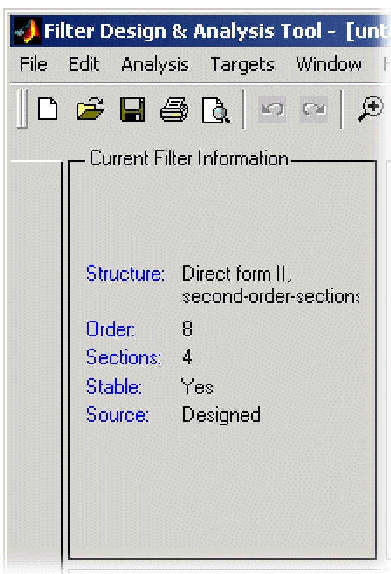
- **Design Filter** — Topic on the Filter Design and Analysis Tool (FDATool) in the Signal Processing Toolbox documentation.
- **Import Filter** — Topic on importing a filter design in the Signal Processing Toolbox documentation.
- **Set Quantization Parameters** — Topic on quantizing filters in the Filter Design and Analysis Tool (FDATool) in the Filter Design Toolbox documentation.

To open a panel, click the appropriate button in the lower-left corner of FDATool.



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 the topic on converting to new filter structures 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 the topic on converting to second-order sections in the Signal Processing Toolbox documentation.

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

Note You may 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 `qfilt` or `dfilt` filter object with the desired structure, and then importing the filter into FDATool. (To learn more about the **Import Filter** panel, see the topic on importing a filter design in the Signal Processing Toolbox documentation.)

Setting Where to Put the Filter

The Filter Realization Wizard creates a subsystem block that implements the specified filter. Set where the Filter Realization Wizard puts the subsystem block by doing the following:

1 Open the **Realize Model** panel in FDATool by clicking the **Realize Model** button  in the lower-left corner of FDATool.

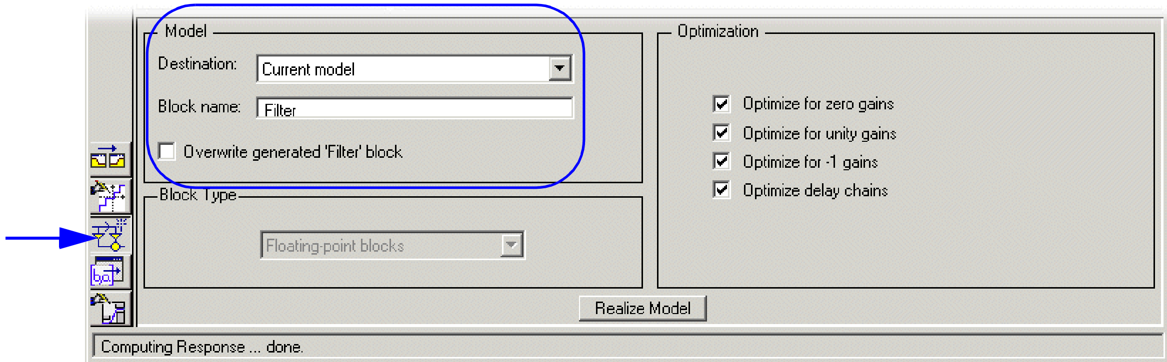
2 Set the **Destination** parameter to one of the following:

- **New model** — Places the subsystem block in a new model.
- **Current model** — Places the subsystem block in the most recently selected model.

3 In the **Block Name** parameter, type a name for the subsystem block.

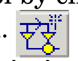
4 If you set the **Destination** parameter to **Current model**, the **Overwrite generated 'Filter' block** parameter becomes enabled, which has the following behavior:

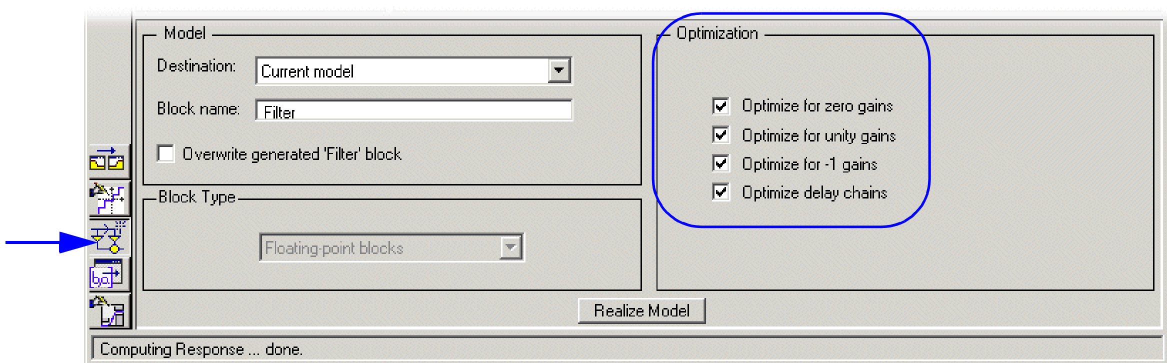
- **Overwrite generated 'Filter' block** ☑ — If there is a previously implemented filter subsystem in the current model that has the name specified in the **Overwrite block** parameter, that subsystem is overwritten by the currently specified filter. If there is no block to overwrite in the current model, the Filter Realization Wizard creates a new filter subsystem with the specified name.
- **Overwrite generated 'Filter' block** ☐ — The Filter Realization Wizard creates a new filter subsystem in the current model without overwriting any previously implemented filters. If there already exists a block in the current model with the name specified in the **Overwrite block** parameter, the Filter Realization Wizard appends the next available number to the new subsystem's name. For example, if you specified the name `myFilter`, and there are blocks in the model named `myFilter`, `myFilter1`, and `myFilter2`, your new filter subsystem is named `myFilter3`.



Optimizing the Filter Structure

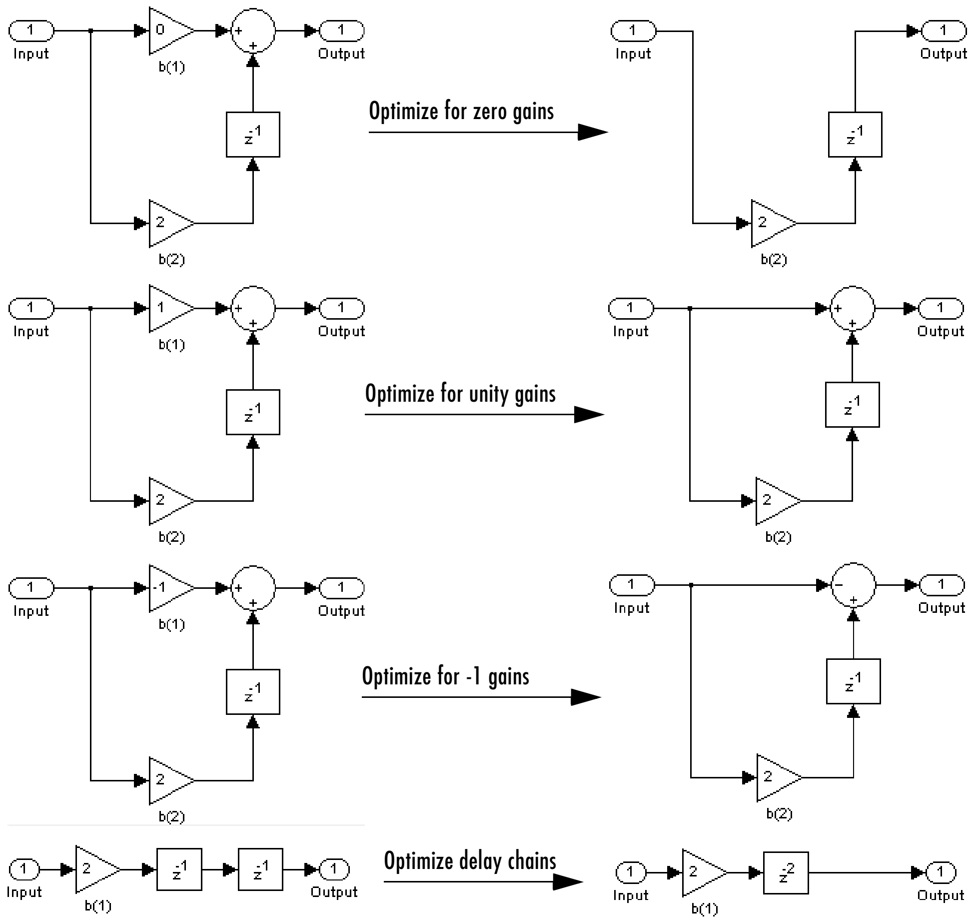
The Filter Realization Wizard creates a subsystem block that implements the specified filter using Sum, Gain, and Integer Delay blocks. To optimize the filter implementation, (for instance, by removing unity gains), do the following:

1. Open the **Realize Model** panel in 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).

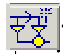
- **Optimize for -1 gains** — Substitute gains equal to -1 with a wire (short circuit), and change the corresponding sums to subtractions.
- **Optimize delay chains** — Substitute any delay chain made up of n unit delays with a single delay by n.

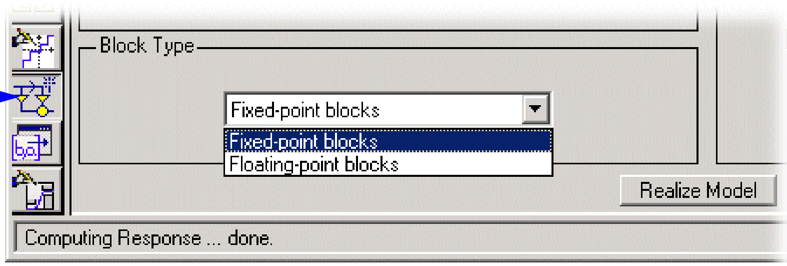


Setting the Data Type of the Filter Implementation

The filter you specify can have various data types for its inputs, outputs, and computations, as described in “Specifying the Filter and Its Data Type Support” on page 7-241.

To ensure that the Filter Realization Wizard's filter implementation accurately reflects the data type(s) you specified for your filter, do the following:

1. Open the **Realize Model** pane by clicking the **Realize Model** button in the lower-left corner of FDATool .
2. Appropriately set the parameter in the **Block Type** region of the **Realize Model** pane (see the following table and figures). The default setting of the parameter depends on the filter, and is always the most appropriate setting for the specified filter.



Appropriate Use of Block Type Settings

Block Type	When to Use	Blocks and Data Types Used in Implementation
Floating-point blocks	Only when implementing purely double-precision filters as described in "Specifying the Filter and Its Data Type Support" on page 7-241.	Sum, Gain, and Integer Delay blocks configured in the specified filter structure, and set to make double-precision computations. See Figure 7-12 on page 7-248.
Fixed-point blocks	When implementing any of the following as described in "Specifying the Filter and Its Data Type Support" on page 7-241: • Fixed-point filters • Single-precision filters • Filters with different input, output, and computation data types	• Sum, Gain, and Integer Delay blocks configured in the specified filter structure, and set to compute using the appropriate data types • Gateway In block for the input quantizer • Conversion blocks for the multiplicand quantizers, and for the output quantizer See the following note and Figure 7-12 on page 7-248.

Note The filter implementation that results from the **Fixed-point blocks** setting described above contains blocks from the Fixed-Point Blockset, which you must install to run the filter in simulations. You can still edit the blocks used to implement the filter without installing the Fixed-Point Blockset. For more information, see the topic on licensing information in the Fixed-Point Blockset documentation.

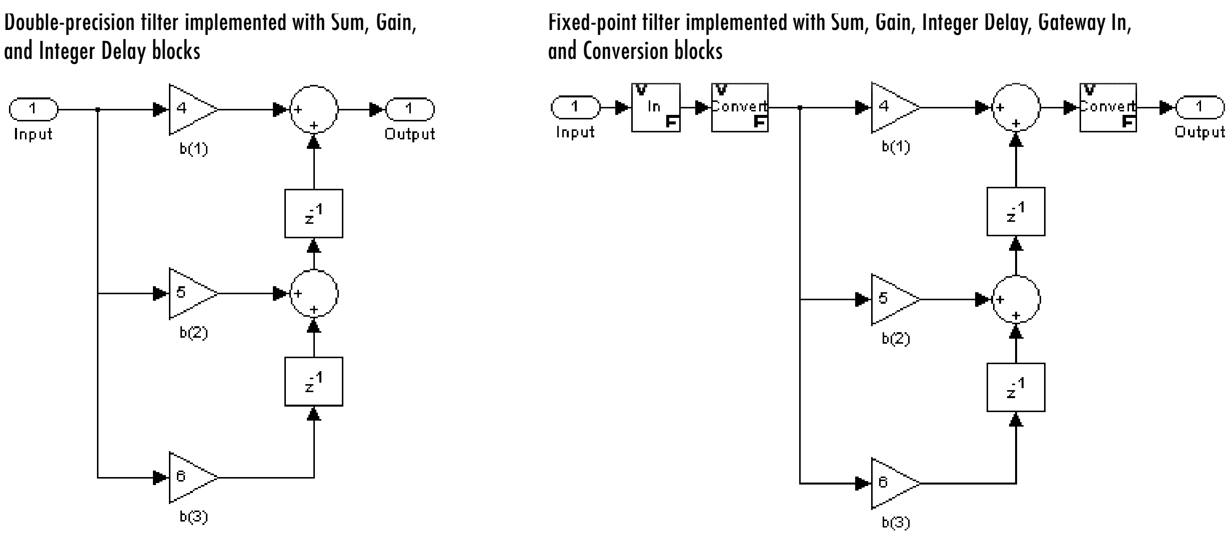


Figure 7-12: Implementations of Double-Precision and Fixed-Point Filters

Corresponding Method for dfilt and qfilt

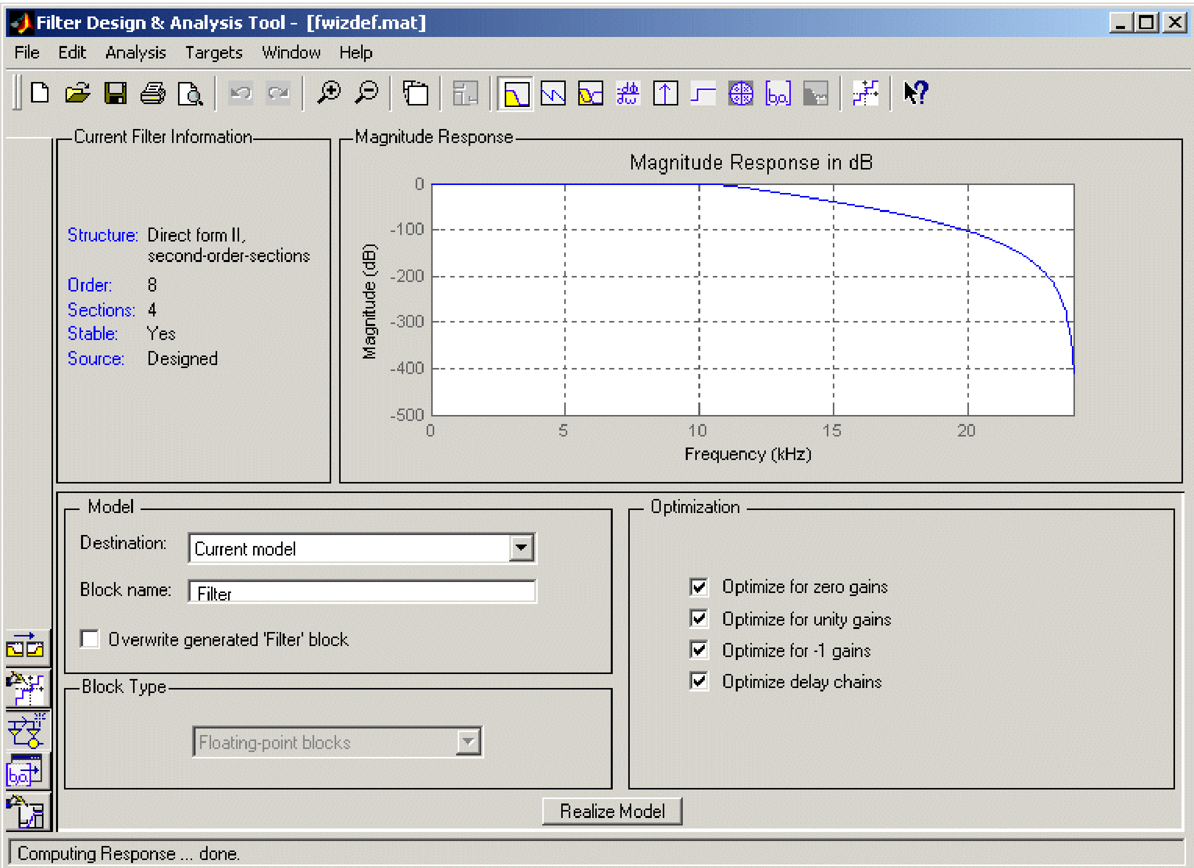
The `dfilt` (digital filter) object in Signal Processing Toolbox and the `qfilt` (quantized filter) object in Filter Design Toolbox both have a method, `realizemdl`, that allows you to access the capabilities of the Filter Realization Wizard from the command line.

For more information about the `realizemdl` method, see the following topics:

- 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

Dialog Box

Note The following parameters for the Filter Realization Wizard are in the **Realize Model** panel 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 7-241.



Destination

The location where the new filter block should be created: in a new model, or in the current (most recently selected) model.

Block name

The name of the new filter block.

Overwrite generated 'Filter' block

When selected, the block overwrites any filter block in the current model with the name specified in the **Block name** parameter. Enabled when the **Destination** parameter is set to **Current model**. For more information, see "Setting Where to Put the Filter" on page 7-244.

Block Type

Determines the data type support of the filter implementation. Set to **Floating-point blocks** when implementing purely double-precision filters. Otherwise, set to **Fixed-point blocks**. For more information, see "Setting the Data Type of the Filter Implementation" on page 7-246.

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 7-245.

Optimize for unity gains

When selected, the block substitutes gains equal to one with a wire (short circuit). For an example, see "Optimizing the Filter Structure" on page 7-245.

Optimize for -1 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 7-245.

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 7-245.

Realize Model

Click to create a subsystem block that implements the specified filter using Sum, Gain, and Integer 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 7-241.

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 Fixed-Point Blockset
- Fixed-point — Supported only when you install the Filter Design Toolbox and Fixed-Point Blockset

To learn how to convert to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

Digital Filter	DSP Blockset
Digital Filter Design	DSP Blockset
`qfilt`	Filter Design Toolbox
`filter`	Filter Design Toolbox
`realizemdl`	Filter Design Toolbox
`dfilt`	Signal Processing Toolbox
`filter`	Signal Processing Toolbox

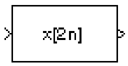
- "Filtering" on page 7-6 — List of all DSP Blockset filtering blocks
- "Filters" on page 4-1 — Examples of when and how to use DSP Blockset filtering blocks
- "Choosing Between Digital Filter Design Block and Filter Realization Wizard" on page 4-4

FIR Decimation

Purpose Filter and downsample an input signal.

Library Filtering / Multirate Filters

Description



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 II transpose 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. The output of the decimator is the first phase of the polyphase filter.

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 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.

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.

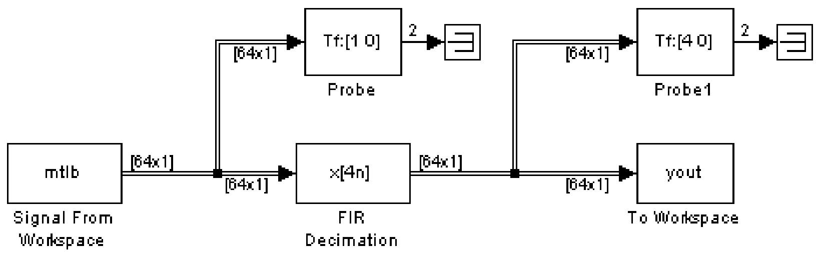
Frame-Based Operation

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 slower (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.

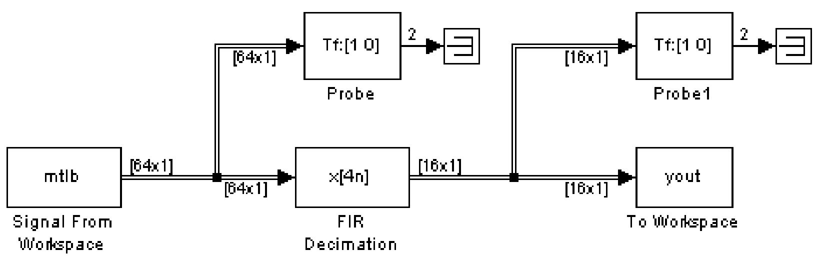
  The example below shows a single-channel input with a frame period of 1 second (**Sample time** = `1/64` and **Samples per frame** = `64` in the Signal From Workspace block) being decimated by a factor of 4 to a frame period of 4 seconds. The input and output frame sizes are identical.



- **Maintain input frame rate**

  The block generates the output at the slower (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 above table. The amount of latency for multirate operation depends on the Simulink tasking mode and the block's sampling mode, as shown in the table below.

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.

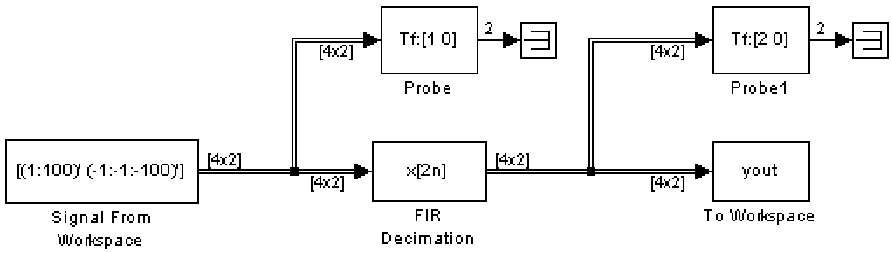
In cases of *one-frame latency*, the first M_i output rows contain zeros, where M_i is the input frame size. The first filtered input sample (first filtered row of the input matrix) appears in the output as sample M_i+1, followed by filtered input samples K+1, 2K+1, and so on. See the example below for an illustration of this case.

See “Excess Algorithmic Delay (Tasking Latency)” on page 3-91 and “The Simulation Parameters Dialog Box” in the Simulink documentation for more information about block rates and the Simulink tasking modes.

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, select **Fixed-step** and **discrete** from the **Type** controls in the **Solver** panel of the **Simulation Parameters** dialog box, and select **MultiTasking** from the **Mode** parameter. 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.

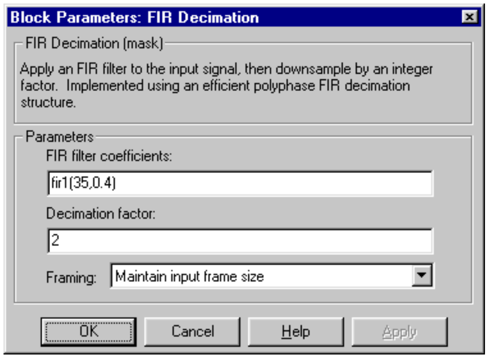
```
yout =

         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 we ran this frame-based multirate model in multitasking mode, the first four (M_i) output rows are zero. The first filtered input matrix row appears in the output as sample 5 (i.e., sample M_i+1).

Example 2

The `dspmrf_menu` demo illustrates the use of the FIR Decimation block in a number of multistage multirate filters.

Dialog Box



FIR filter coefficients

The lowpass FIR filter coefficients, in descending powers of z.

Decimation factor

The integer factor, K, by which to decrease the sample rate of the input sequence.

Framing

For frame-based operation, the method by which to implement the decimation; reduce the output frame rate, or reduce the output frame size.

Supported Data Types

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

To learn how to convert to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

Downsample	DSP Blockset
FIR Interpolation	DSP Blockset
FIR Rate Conversion	DSP Blockset
`decimate`	Signal Processing Toolbox
`fir1`	Signal Processing Toolbox
`fir2`	Signal Processing Toolbox
`firls`	Signal Processing Toolbox
`remez`	Signal Processing Toolbox

See the following sections for related information:

- "Converting Sample Rates and Frame Rates" on page 3-20
- "Multirate Filters" on page 4-32
- "Filtering" on page 7-6 — List of all DSP Blockset filtering blocks

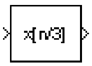
Purpose

Upsample and filter an input signal.

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:

- 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 II transpose 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.

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 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.

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.

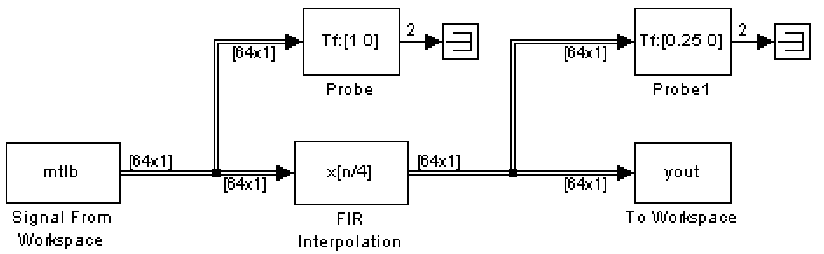
Frame-Based Operation

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 faster (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.

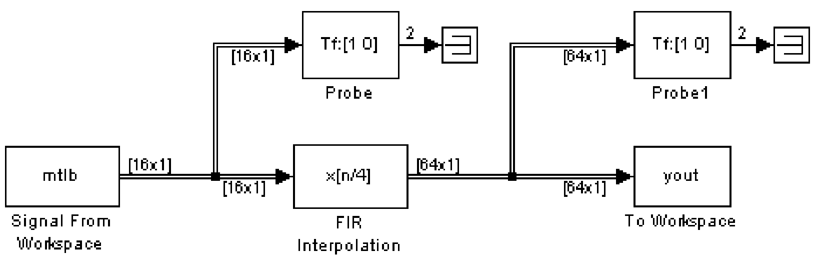
  The example below shows a single-channel input with a frame period of 1 second (**Sample time** = 1/64 and **Samples per frame** = 64 in the Signal From Workspace block) being interpolated by a factor of 4 to a frame period of 0.25 second. The input and output frame sizes are identical.



- **Maintain input frame rate**

  The block generates the output at the faster (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 above table. The amount of latency for multirate operation depends on the Simulink tasking mode and the block's sampling mode, as shown in the table below.

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, followed immediately by the first filtered input sample, L-1 interpolated values, and so on.

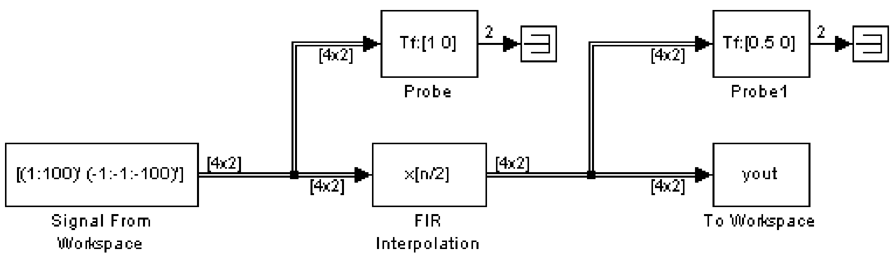
In cases of *one-frame latency*, the first M_i output rows contain zeros, where M_i is the input frame size. The first filtered input sample (first filtered row of the input matrix) appears in the output as sample M_i+1, followed by L-1 interpolated values, the second filtered input sample, and so on. See the example below for an illustration of this case.

See "Excess Algorithmic Delay (Tasking Latency)" on page 3-91 and "The Simulation Parameters Dialog Box" in the Simulink documentation for more information about block rates and the Simulink tasking modes.

Example

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, select **Fixed-step** and **discrete** from the **Type** controls in the **Solver** panel of the **Simulation Parameters** dialog box, and select **MultiTasking** from the **Mode** parameter. 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.

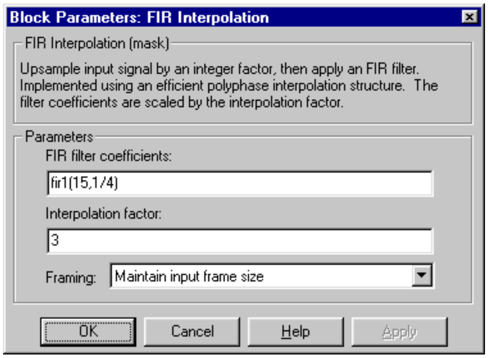
```
yout =

         0         0
         0         0
         0         0
         0         0
    0.0386   -0.0386
    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 four (M_i) output rows are zero. The first filtered input matrix row appears in the output as sample 5 (i.e., sample M_i+1). Every second row is an interpolated value.

Example 2

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.

Dialog Box



FIR filter coefficients

The FIR filter coefficients, in descending powers of z.

Interpolation factor

The integer factor, L, by which to increase the sample rate of the input sequence.

Framing

For frame-based operation, the method by which to implement the interpolation: increase the output frame rate, or increase the output frame size.

Supported Data Types

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

To learn how to convert to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

FIR Decimation	DSP Blockset
FIR Rate Conversion	DSP Blockset
Upsample	DSP Blockset
`fir1`	Signal Processing Toolbox
`fir2`	Signal Processing Toolbox
`firls`	Signal Processing Toolbox
`interp`	Signal Processing Toolbox
`remez`	Signal Processing Toolbox

See the following sections for related information:

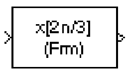
- “Converting Sample Rates and Frame Rates” on page 3-20
- “Multirate Filters” on page 4-32
- “Filtering” on page 7-6 — List of all DSP Blockset filtering blocks

FIR Rate Conversion

Purpose Upsample, filter, and downsample an input signal.

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:



- The block upsamples the input to a higher rate by inserting L-1 zeros between input samples.
- The upsampled data is passed through a direct-form II transpose FIR filter.
- 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. The output of the interpolator is the first filter phase, while the output of the decimator is the last filter phase. When both K and L are greater than 1, the resulting output is the last decimation phase from the first interpolation phase.

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 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.

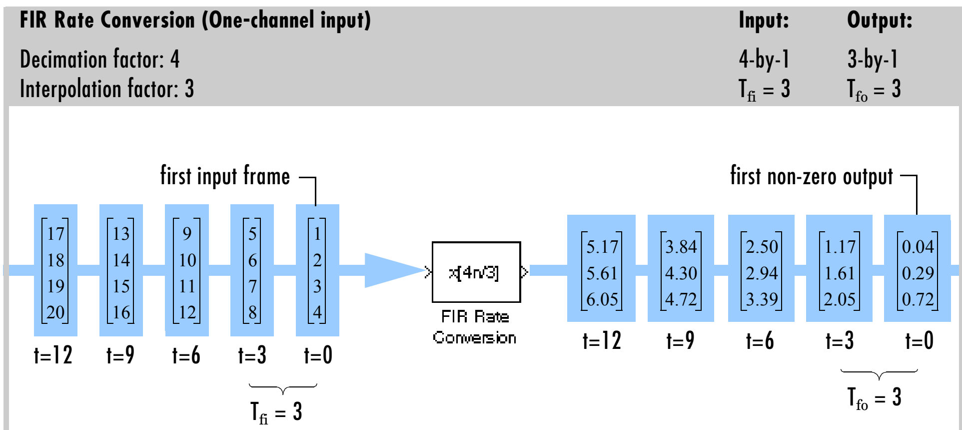
Frame-Based Operation

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.

Examples

The `dspsrcnv` demo compares sample rate conversion performed by the FIR Rate Conversion block with the same conversion performed by a cascade of Upsample, Digital Filter, and Downsample blocks.

Diagnostics

An error is generated if the relation between K and L shown above is not satisfied.

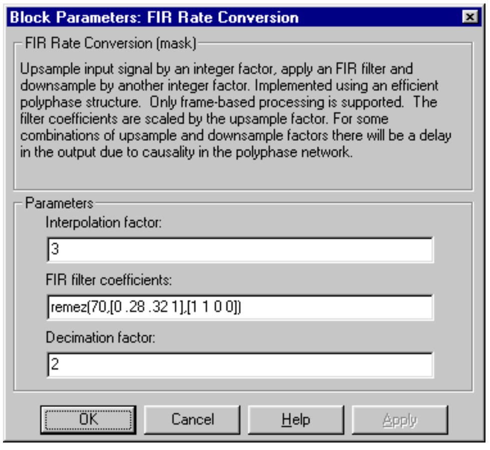
```
(Input port width)/(Output port width) must equal the
(Decimation factor)/(Interpolation factor).
```

A warning is generated if L and K are not relatively prime; that is, if 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.

Dialog Box



Interpolation factor
: The integer factor, L, by which to upsample the signal before filtering.

FIR filter coefficients
: The FIR filter coefficients, in descending powers of z.

Decimation factor
: The integer factor, K, by which to downsample the signal after filtering.

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

To learn how to convert to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

Downsample	DSP Blockset
FIR Decimation	DSP Blockset
FIR Interpolation	DSP Blockset
Upsample	DSP Blockset
`fir1`	Signal Processing Toolbox
`fir2`	Signal Processing Toolbox
`firls`	Signal Processing Toolbox
`remez`	Signal Processing Toolbox
`upfirdn`	Signal Processing Toolbox

See the following sections for related information:

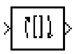
- "Converting Sample Rates and Frame Rates" on page 3-20
- "Multirate Filters" on page 4-32
- "Filtering" on page 7-6 — List of all DSP Blockset filtering blocks

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 **Columns** is selected 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
```

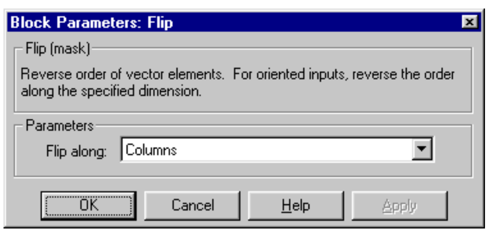
For convenience, length-M 1-D vector inputs are treated as M-by-1 column vectors for vertical flipping.

When **Rows** is selected 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.

Dialog Box



Flip along

The dimension along which to flip the input. **Columns** specifies vertical flipping, while **Rows** specifies horizontal flipping.

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 to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

Selector	Simulink
Transpose	DSP Blockset
Variable Selector	DSP Blockset
`flipud`	MATLAB
`fliplr`	MATLAB

Also see "Indexing" on page 7-14 for a list of all the blocks in the Indexing library.

Forward Substitution

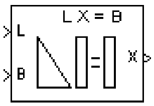
Purpose

Solve the equation L*X*=B for *X* when L is a lower triangular matrix.

Library

Math Functions / Matrices and Linear Algebra / Linear System Solvers

Description

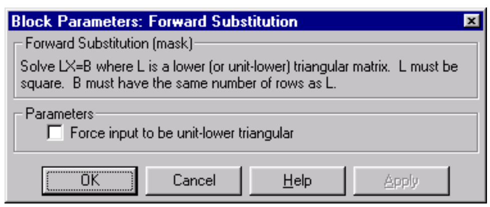


The Forward Substitution block solves the linear system L*X*=B by simple forward substitution of variables, where L is the lower triangular M-by-M matrix input to the `L` port, and B is the M-by-N matrix input to the `B` port. The output is the solution of the equations, the M-by-N matrix *X,* and is 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 **Force input to be unit-lower triangular** is selected, the block replaces the elements on the diagonal of L with ones. 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 1s when selected. Tunable in simulation.

Supported Data Types

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

To learn how to convert to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Autocorrelation LPC	DSP Blockset
Cholesky Solver	DSP Blockset
LDL Solver	DSP Blockset
Levinson-Durbin	DSP Blockset
LU Solver	DSP Blockset
QR Solver	DSP Blockset

See “Solving Linear Systems” on page 6-7 for related information. Also see “Linear System Solvers” on page 7-9 for a list of all the blocks in the Linear System Solvers library.

Frame Status Conversion

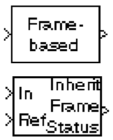
Purpose

Specify the frame status of the output, sample-based or frame-based.

Library

Signal Management / Signal Attributes

Description

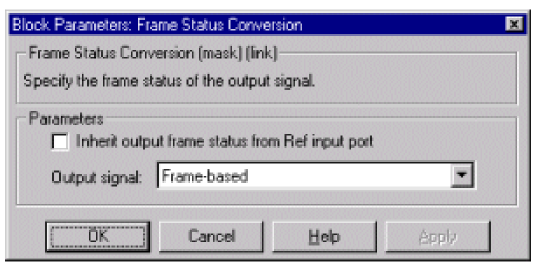


The Frame Status Conversion block passes the input through to the output, and sets the output frame status to the **Output signal** parameter, which can be either **Frame-based** or **Sample-based**. The output frame status can also be inherited from the signal at the `Ref` (reference) input port, which is made visible by selecting the **Inherit output frame status from Ref input port** check box.

If 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, if 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 (i.e., a single channel).

If 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

- 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 to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Check Signal Attributes	DSP Blockset
Convert 1-D to 2-D	DSP Blockset
Convert 2-D to 1-D	DSP Blockset
Inherit Complexity	DSP Blockset

Also see “Signal Attributes” on page 7-15 for a list of all the blocks in the Signal Attributes library.

From Wave Device

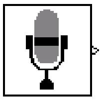
Purpose

Read audio data from a 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.

If the audio source contains two channels (stereo), the **Stereo** check box should be selected. If 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 PCM (pulse code modulation) 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:

- **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)

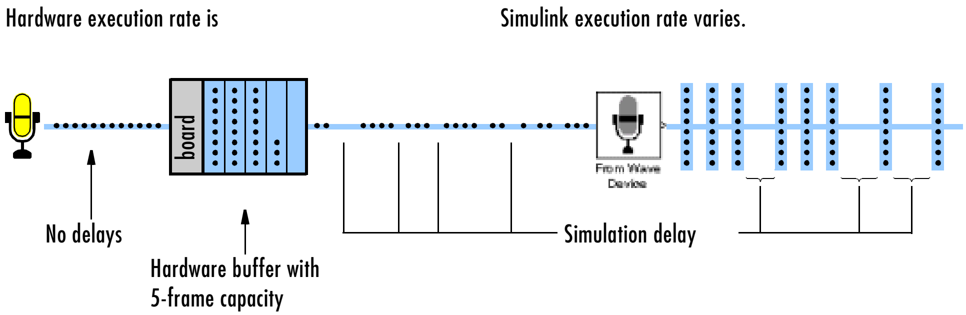
Higher sample width settings require more memory but yield better fidelity. The output from the block is independent of the **Sample width (bits)** setting. The output data type is determined by the **Data type** parameter setting.

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 in principle 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.)



If 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

If the buffer size is too small in relation to the simulation throughput rate, the buffer may 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 may increase model latency.

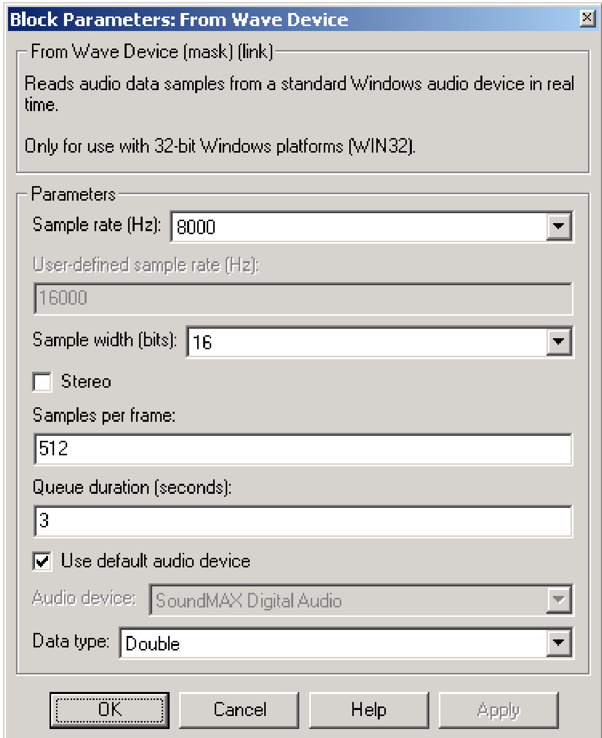
- *Increase the simulation throughput rate*

  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 3-85 of this book, and “Improving Simulation Performance and Accuracy” in the 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 checked, mono (one-channel) inputs when unchecked. Stereo output is M-by-2; mono output is M-by-1.

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** if 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 data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

From Wave File	DSP Blockset
To Wave Device	DSP Blockset
`audiorecorder`	MATLAB
`audiodevinfo`	MATLAB

See "Importing WAV Files" on page 3-71 for related information. Also see "Windows (WIN32)" on page 7-13 for a list of all the blocks in the Windows (WIN32) library.

From Wave File

Purpose

Read audio data from a Microsoft Wave (`.wav`) file. (32-bit Windows operating systems only)

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.

Output Data Type	Output Amplitude Range
`double`	±1
`single`	±1
`int16`	-32768 to 32767 (-2^{15} to $2^{15} - 1$)
`uint8`	0 to 255

The audio data must be in uncompressed PCM (pulse code modulation) 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. If 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 in either case.

If 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. If 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 frame** parameter. For M=1, the output is sample-based; otherwise, the output is frame-based.

The output frame period, T_{fo}, is

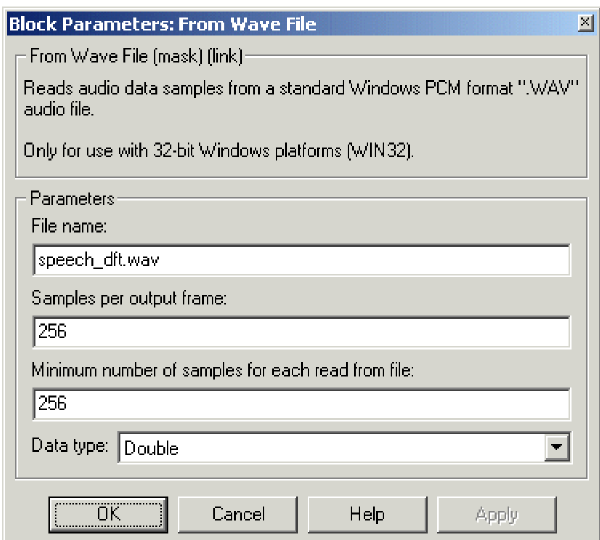
$$T_{fo} = \frac{M}{F_s},$$

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.

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.

Dialog Box



File name

The path and name of the file to read. Paths can be relative or absolute.

Samples per output frame

The number of samples in each output frame, M.

Minimum number of samples for each read from file

The number of consecutive samples to acquire from the file with each file access, L.

Data Type

The output data type: double-precision, single-precision, signed 16-bit integer, or unsigned 8-bit integer. The data type setting determines the output's amplitude range, as shown in the table above.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- 16-bit signed integer
- 8-bit unsigned integer

To learn how to convert data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

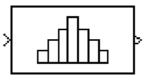
From Wave Device	DSP Blockset
Signal From Workspace	DSP Blockset
To Wave File	DSP Blockset
`wavread`	MATLAB

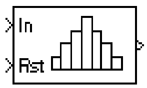
See "Importing WAV Files" on page 3-71 for related information. Also see "Windows (WIN32)" on page 7-13 for a list of all the blocks in the Windows (WIN32) library.

Purpose Generate the histogram of an 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 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 their magnitudes.

The histogram value for a given bin represents the *frequency of occurrence* of the input values bracketed by that bin. The upper-boundary of the highest-valued bin is specified by the **Maximum value of input** parameter, B_M, and the lower-boundary of the lowest-valued bin is specified by 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.

Basic Operation

When the **Running histogram** check box is *not* selected, 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 the **Normalized** check box is selected, the block scales each column of the output so that `sum(y(:,j))` is 1.

Running Operation

When the **Running histogram** check box is selected, 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.

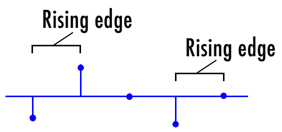
The output is a sample-based n-by-N matrix whose jth column reflects the current state of the jth histogram. The block resets the running histogram (by emptying all bins of all histograms) when it detects a reset event at the optional `Rst` port, as described next.

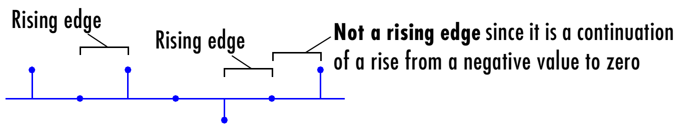
Resetting the Running Histogram. The block resets the running histogram whenever a reset event is detected at the optional `Rst` port. The reset signal rate must be a positive integer multiple of the rate of the data signal input.

To enable the `Rst` port, select the **Reset port** parameter. The reset event is specified by the **Trigger type** parameter, and can be one of the following:

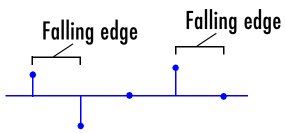
- **Rising edge** — Triggers a reset operation when the `Rst` input does one of the following:
  - Rises from a negative value to a positive value or zero

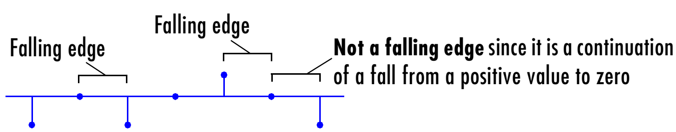
- Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)





- **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.

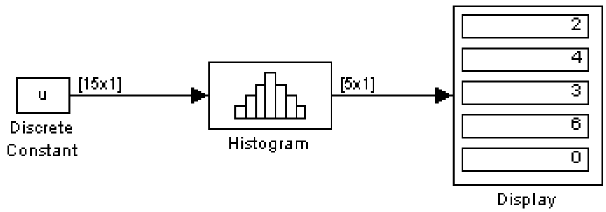
Note When running simulations in the Simulink **MultiTasking** mode, sample-based reset signals have a one-sample latency, and frame-based reset signals have one frame of latency. Thus, there is a one-sample or one-frame delay between the time the block detects a reset event, and when it applies the reset. For more information on latency and the Simulink tasking modes, see

"Excess Algorithmic Delay (Tasking Latency)" on page 3-91 and the topic on the **Simulation Parameters** dialog box in the Simulink documentation.

Example

The model below illustrates the Histogram block's basic operation for a single-channel input, u, where

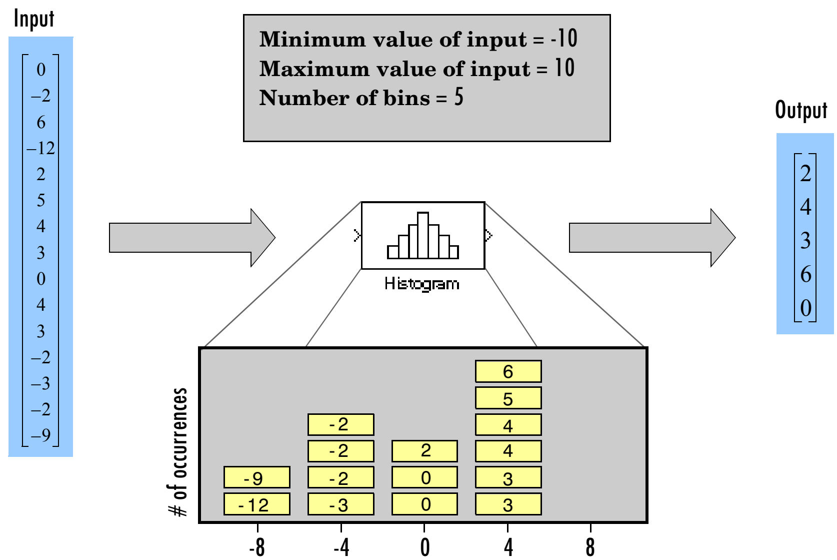
```
u = [0 -2 6 -12 2 5 4 3 0 4 3 -2 -3 -2 -9]'
```



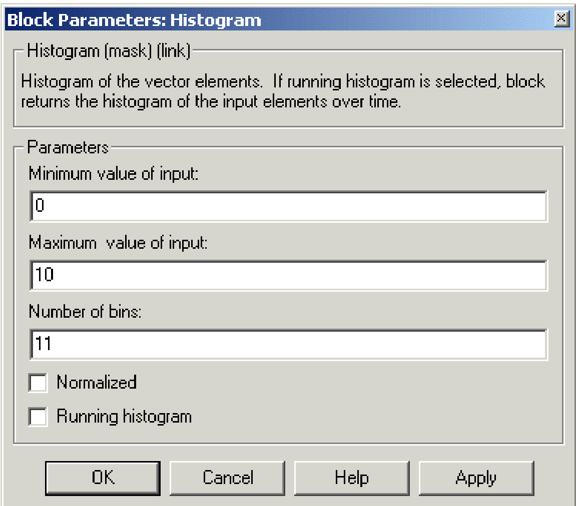
The parameter settings for the Histogram block are:

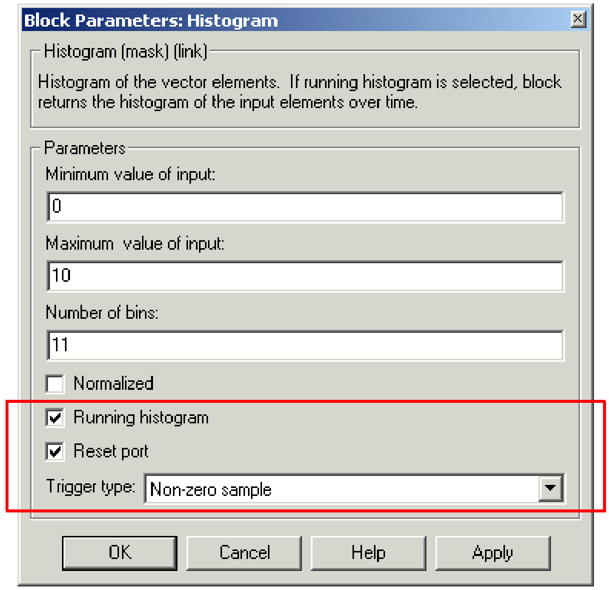
- **Minimum value of input** = -10
- **Maximum value of input** = 10
- **Number of bins** = 5
- **Normalized** = ☐
- **Running histogram** = ☐

The resulting bin width is 4, as shown below.


Input
0
−2
6
−12
2
5
4
3
0
4
3
−2
−3
−2
−9
Minimum value of input = -10
Maximum value of input = 10
Number of bins = 5
Histogram
Output
2
4
3
6
0
# of occurrences
-9
-12
-2
-2
-2
-3
2
0
0
6
5
4
4
3
3
-8
-4
0
4
8

Dialog Box





Minimum value of input

The lower boundary, B_m, of the lowest-valued bin. Tunable.

Maximum value of input

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. Tunable.

Enables running operation when selected.

Running histogram

Set to enable the running histogram operation, and clear to enable basic histogram operation. For more information, see “Basic Operation” on page 7-285 and “Running Operation” on page 7-286.

Reset port

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. This parameter is enabled only when you set the **Running histogram** parameter. For more information, see “Running Operation” on page 7-286.

Trigger type

The type of event that resets the running histogram. For more information, see “Resetting the Running Histogram” on page 7-286. This parameter is enabled only when you set the **Reset port** parameter.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Boolean — The block accepts Boolean inputs to the `Rst` port, which is enabled when you set the **Reset port** parameter.

To learn how to convert to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Sort	DSP Blockset
`hist`	MATLAB

Also see “Statistics” on page 7-18 for a list of all the blocks in the Statistics library.

IDCT

Purpose

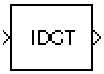
Compute the IDCT of the input.

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 lth column contains the length-M IDCT of the corresponding input column.

$$y(m,l) = \sum_{k=1}^{M} w(k)u(k,l)\cos\frac{\pi(2m-1)(k-1)}{2M}, \qquad m = 1, \ldots, M$$

where

$$w(k) = \begin{cases} \dfrac{1}{\sqrt{M}}, & k = 1 \\ \sqrt{\dfrac{2}{M}}, & 2 \le k \le M \end{cases}$$

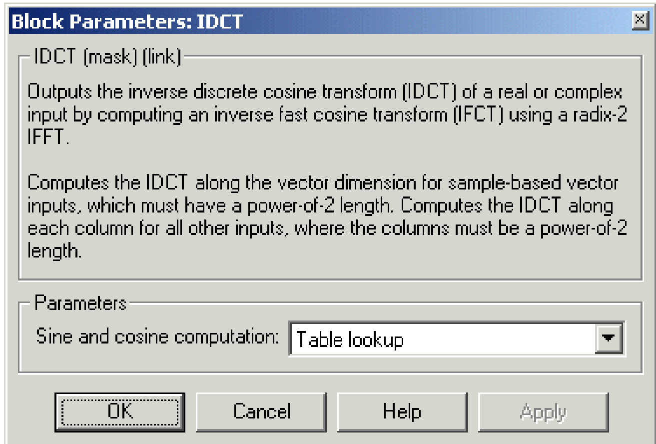
The output is always frame-based, and the output sample rate and data type (real/complex) are the same as those of the input.

For convenience, length-M 1-D vector inputs and *sample-based* length-M row vector inputs are processed as single channels (i.e., 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 in the IFFT and fast IDCT algorithms used to compute the IDCT. 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.

Dialog Box



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.

Supported Data Types

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

To learn how to convert to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

DCT	DSP Blockset
IFFT	DSP Blockset
`idct`	Signal Processing Toolbox

Also see "Transforms" on page 7-19 for a list of all the blocks in the Transforms library.

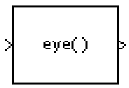
Purpose

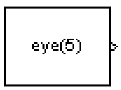
Generate a matrix with ones on the main diagonal and zeros elsewhere.

Library

- DSP 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 the **Inherit output port attributes from input port** check box is selected, 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
```

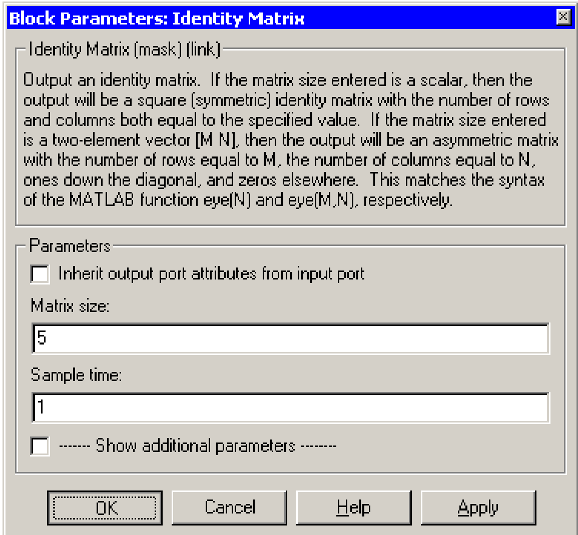
When the **Inherit output port attributes from input port** check box is *not* selected, the input port is disabled, and the dimensions of the output matrix are determined by the **Matrix size** parameter. A scalar value, `M`, specifies an M-by-M identity matrix, while a two-element vector, `[M N]`, specifies an M-by-N unit-diagonal matrix. The output is sample-based, and has the sample period specified by the **Sample time** parameter.

Example

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



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

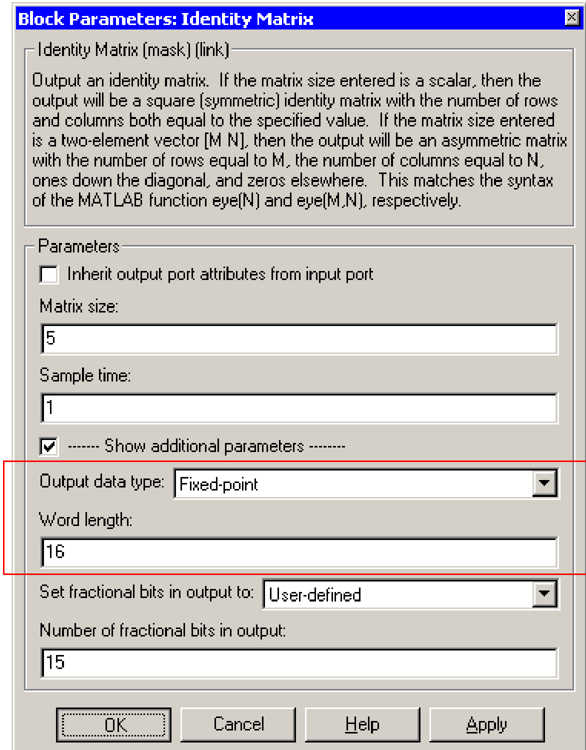
The number of rows and columns in the output matrix: a scalar `M` for a square M-by-M output, or a vector `[M N]` for an M-by-N output. This parameter is disabled when **Inherit input port attributes from input port** is selected.

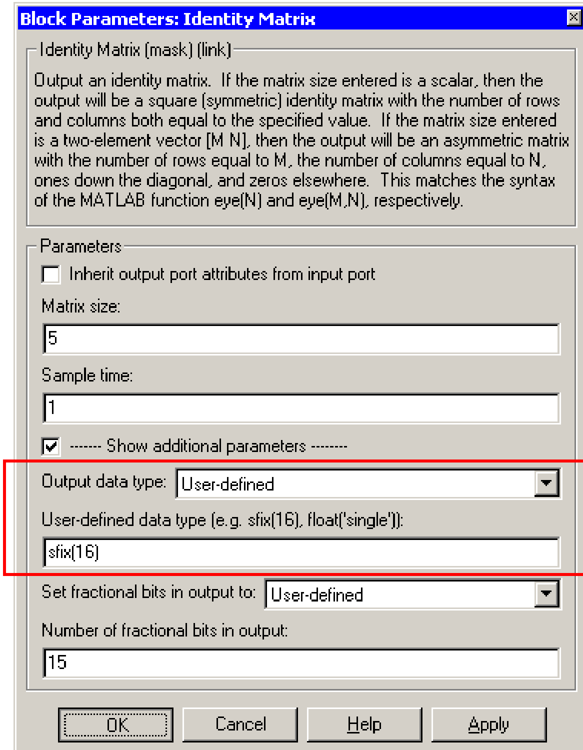
Sample time

The discrete sample period of the output. This parameter is disabled when **Inherit input port attributes from input port** is selected.

Show additional parameters

If selected, additional parameters specific to implementation of the block become visible as shown.





Output data type

Specify the output data type in one of the following ways:

- Choose one of the built-in data types from the drop-down list.
- Choose **Fixed-point** to specify the output data type and scaling in the **Word length**, **Set fractional bits in output to,** and **Number of fractional bits in output** parameters.
- Choose **User-defined** to specify the output data type and scaling in the **User-defined data type**, **Set fractional bits in output to,** and **Number of fractional bits in output** 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 if **Fixed-point** is selected 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 the Fixed-Point Blockset. This parameter is only visible if **User-defined** is selected for the **Output data type** parameter.

Set fractional bits 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 **Number of fractional bits in output** parameter.

This parameter is only visible if **Fixed-point** or **User-defined** is selected for the **Output data type** parameter, and if the specified output data type is a fixed-point data type.

Number of fractional bits in output

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 if **Fixed-point** or **User-defined** is selected for the **Output data type** parameter, and if **User-defined** is selected for the **Set fractional bits 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 to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Constant Diagonal Matrix	DSP Blockset
DSP Constant	DSP Blockset
`eye`	MATLAB

Also see the following topics:

- “Creating Signals Using Constant Blocks” on page 3-33 — How to use this and other blocks to generate constant signals
- “DSP Sources” on page 7-3 — List of all blocks in the DSP Sources library
- “Matrix Operations” on page 7-11 — List of all the blocks in the Matrix Operations library.

Purpose Compute the inverse discrete wavelet transform (IDWT) of the input signal

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 block reference 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 the Dyadic Synthesis Filter Bank block reference.

Examples See "Examples" on page 7-214 in the Dyadic Synthesis Filter Bank reference page.

See Also

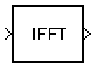
Dyadic Synthesis Filter Bank	DSP Blockset

Also see "Transforms" on page 7-19 for a list of all the blocks in the Transforms library.

Purpose Compute the IFFT of the input.

Library Transforms

Description



The IFFT block computes the inverse fast Fourier transform (IFFT) of each channel in the 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 is always frame-based, and each output column contains the M-point inverse discrete Fourier transform (IDFT) of the corresponding input channel.

```
y = ifft(u,M)      % Equivalent MATLAB code
```

The kth entry of the lth output channel, $y(k, l)$, is the kth point of the M-point IDFT of the lth input channel.

$$y(k,l) = \frac{1}{M}\sum_{m=1}^{M} u(m,l)e^{j2\pi(m-1)(k-1)/M} \qquad k = 1, \ldots, M \tag{7-3}$$

You can choose to output a scaled version of the input's IDFT, $M \cdot y(k, l)$, by setting the parameter **Skip normalization by transform length, N**.

$$M \cdot y(k,l) = \sum_{m=1}^{M} u(m,l)e^{j2\pi(m-1)(k-1)/M} \qquad k = 1, \ldots, M \tag{7-4}$$

For information on block output characteristics and how to configure the block computation methods, see other sections of this reference page.

Sections of This Reference Page

Input and Output Characteristics

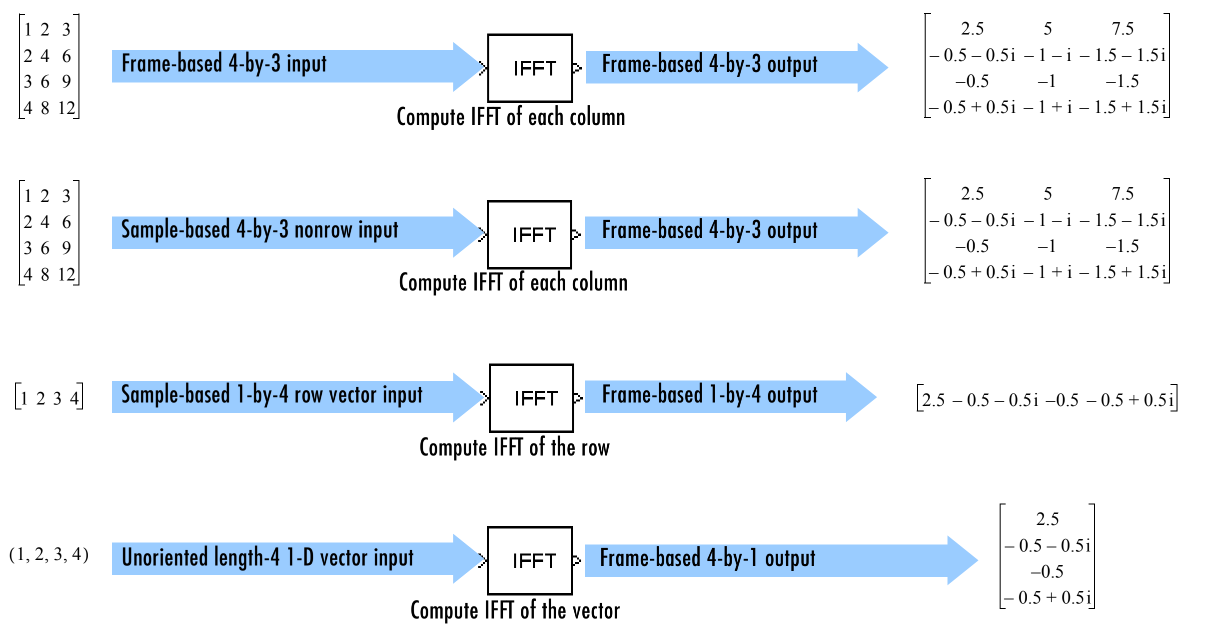
The following table describes all valid block input types, their corresponding outputs, and the dimension along which the block computes the IDFT.

Note For M-by-N and length-M inputs, 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. To get valid outputs, you must correctly set the **Input is in bit-reversed order** parameter to indicate the ordering of the block input. When the input is conjugate symmetric and you want to get a real-valued output, set the **Input is conjugate symmetric** parameter.

Valid Block Inputs • Must be 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 unless you set the **Input is conjugate symmetric** parameter when your input is conjugate-symmetric, in which case the output is real-valued • Same dimension as input (for 1-D inputs, output is a length-M column) • Each column (each row for sample-based row inputs) contains the M-point IDFT of the corresponding input channel in linear order. If the parameter **Skip normalization by transform length, N** is set, rather than computing the IDFT, the block computes a scaled version of the IDFT given by Equation 7-4.
Sample-based M-by-N matrix, $M \neq 1$	Column	
Sample-based 1-by-M *row vector*	Row	
1-D length-M vector	Vector	

Click here in the MATLAB Help Browser to open a Simulink model based on the following diagram.

Effects of Block Input Size, Dimension, and Frame Status on Block Output



Conjugate Symmetric Input

When the block input is conjugate symmetric and you want real-valued outputs, set the **Input is conjugate symmetric** parameter, which also optimizes the block's computation method. A common source of conjugate symmetric data is the FFT block, which yields conjugate symmetric output when its input is real-valued.

Note If the IFFT block input is conjugate symmetric but you do not set the **Input is conjugate symmetric** parameter, you do not get a real-valued output. Instead, you get a complex-valued output with small imaginary parts.

The block output is invalid if you set this parameter when the input is not conjugate symmetric.

Inputs in Bit-Reversed Order

When the block input is in bit-reversed order, you must set the parameter **Input is in bit-reversed order** to get a valid output. The block output is invalid if you set this parameter when the input is not in bit-reversed order. A common source of bit-reversed inputs is the FFT block, as illustrated in the FFT block example, “Use of Outputs in Bit-Reversed Order” on page 7-235.

For a definition of bit-reversed and linear order, see the FFT block reference page section, “Description of Bit-Reversed Ordering” on page 7-232.

Selecting the Twiddle Factor Computation Method

The FFT block and IFFT block both have a parameter, **Twiddle factor computation**. Setting this parameter in the IFFT block is very similar to setting it in the FFT block. For details, see the FFT block reference page section, “Selecting the Twiddle Factor Computation Method” on page 7-233.

Optimizing the Table of Trigonometric Values

The FFT block and IFFT block both have a parameter, **Optimize table for**. Setting this parameter in the IFFT block is very similar to setting it in the FFT block. For details, see the FFT block reference page section, “Optimizing the Table of Trigonometric Values” on page 7-234.

Algorithms Used for IFFT Computation

Depending on whether the block input is real- or complex-valued and conjugate symmetric, the block uses one or more of the following algorithms as summarized in the next table:

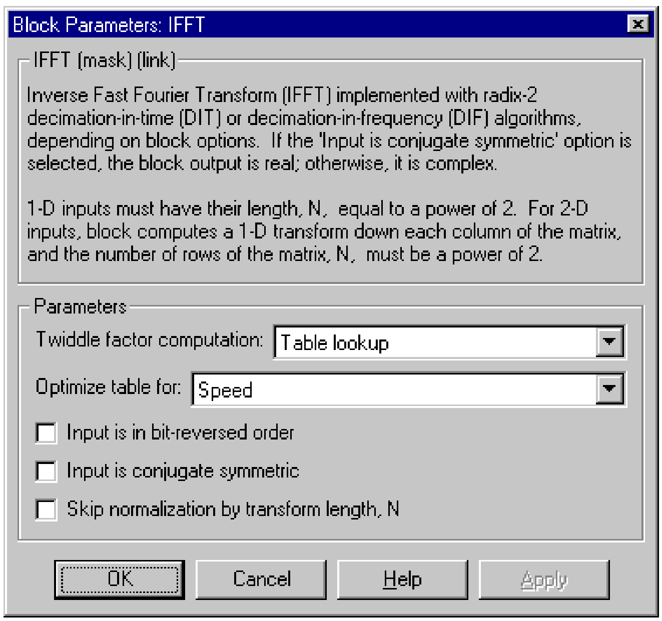
- Radix-2 decimation-in-time (DIT) algorithm
- Half-length algorithm
- Double-signal algorithm

Input Complexity	Other Parameter Settings	Algorithms Used for FFT Computation
Complex	not applicable	Radix-2 DIT
Real	☐ Input is conjugate symmetric	Radix-2 DIT
Real	☑ Input is conjugate symmetric	Radix-2 DIT in conjunction with the half-length and double-signal algorithms when possible

Example

For an example of how to optimize computations when using both the IFFT block and FFT block in the same model, see the FFT block reference page example, “Use of Outputs in Bit-Reversed Order” on page 7-235.

Dialog Box



Twiddle factor computation

Computation method of the term $e^{j2\pi(m-1)(k-1)/M}$ in Equation 7-3. 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.

Optimize table for

Optimization of the table of sine and cosine values for **Speed** or **Memory**. Active only when **Twiddle factor computation** is set to **Table lookup**.

Input is in bit-reversed order

Set when the input is in bit-reversed order, and clear when the input is in linear order. The block yields invalid outputs if you do not set this parameter correctly.

Input is conjugate symmetric

Set when the block input is conjugate symmetric and you want real-valued outputs. The block output is invalid if you set this parameter when the input is not conjugate symmetric.

Skip normalization by transform length, N

When set, rather than computing the IDFT, the block computes a scaled version of the IDFT given by Equation 7-4.

Supported Data Types

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

To learn how to convert to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

FFT	DSP Blockset
IDCT	DSP Blockset
Pad	DSP Blockset
Zero Pad	DSP Blockset
`bitrevorder`	Signal Processing Toolbox
`fft`	Signal Processing Toolbox
`ifft`	Signal Processing Toolbox

Also see “Transforms” on page 7-19 for a list of all the blocks in the Transforms library.

Inherit Complexity

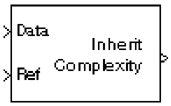
Purpose

Change the complexity of the input to match that of a reference signal.

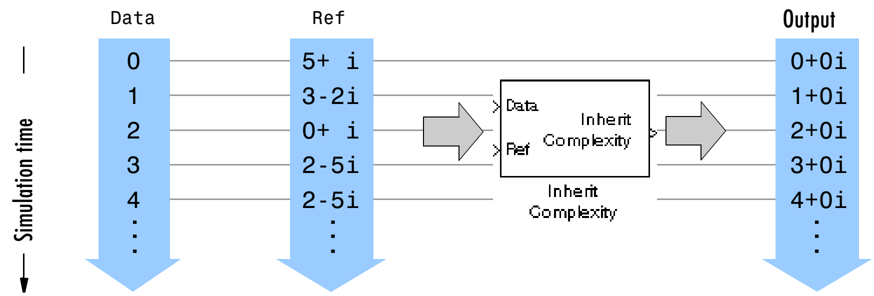
Library

Signal Management / Signal Attributes

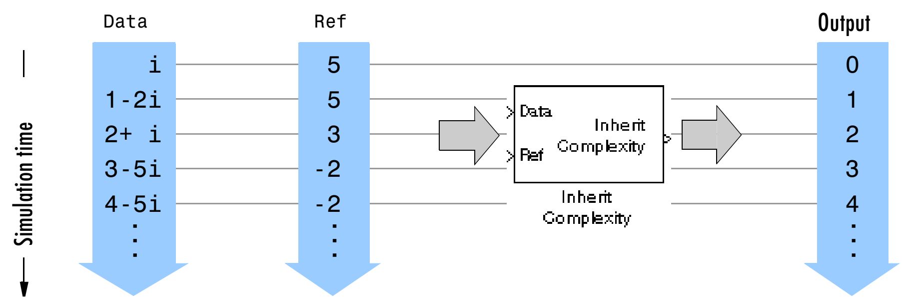
Description



The Inherit Complexity block alters the input data at the `Data` port to match the complexity of the reference input at the `Ref` port. If 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.

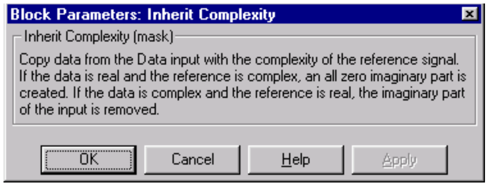


If the `Data` input is complex, and the `Ref` input is real, the block outputs the real component of the `Data` input.



If both the `Data` input and `Ref` input are real, or if both the `Data` input and `Ref` input are complex, the block propagates the `Data` input with no change.

Dialog Box



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 to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Check Signal Attributes	DSP Blockset
Complex to Magnitude-Angle	Simulink
Complex to Real-Imag	Simulink
Magnitude-Angle to Complex	Simulink
Real-Imag to Complex	Simulink

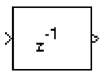
Also see “Signal Attributes” on page 7-15 for a list of all the blocks in the Signal Attributes library.

Integer Delay

Purpose Delay an input by an integer number of sample periods.

Library Signal Operations

Description



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.

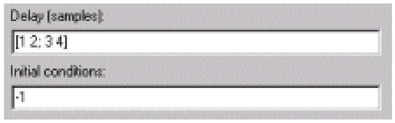
For example, if the input is M-by-1 and v is the matrix `[v(1) v(2) ... v(M)]'`, the first channel is delayed by `v(1)` sample intervals, the second channel is delayed by `v(2)` sample intervals, and so on. Note that when a channel is delayed for Δ sample-time units, the output sample at time t is the input sample at time $t-\Delta$. If $t-\Delta$ 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,

$$\begin{bmatrix} -1 & -1 \\ -1 & -1 \end{bmatrix}, \begin{bmatrix} u_{11}^{1} & -1 \\ -1 & -1 \end{bmatrix}, \begin{bmatrix} u_{11}^{2} & u_{12}^{1} \\ -1 & -1 \end{bmatrix}, \begin{bmatrix} u_{11}^{3} & u_{12}^{2} \\ u_{21}^{1} & -1 \end{bmatrix}, \begin{bmatrix} u_{11}^{4} & u_{12}^{3} \\ u_{21}^{2} & u_{22}^{1} \end{bmatrix}, \ldots$$

where u_{ij}^{k} is the *i,j*th element of the *k*th 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" below.

Initial conditions cannot be specified by full matrices.

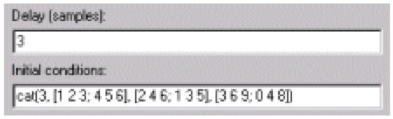
Time-Varying Initial Conditions. A time-varying initial condition in sample-based mode can be specified in one of the following ways:

- *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,

  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 if 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,



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.

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.

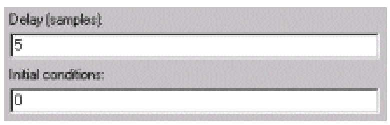
For frame-based inputs, the **Delay** parameter can be a scalar integer by which to equally delay all channels. It can also be a 1-by-N row vector, each element of which serves as the delay for the corresponding channel of the N-channel input. Likewise, it can also be an M-by-1 column vector, each element 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 $t-\Delta$. If $t-\Delta$ 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,

  the first five samples in each of the N channels are zero. Note that if 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” below 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,

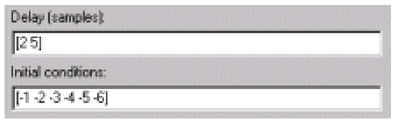
    `D = d + 1`

    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,

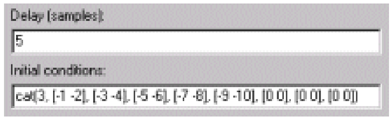


the block outputs the following sequence of frames at the start of the simulation.

$$\begin{bmatrix} -4 & -1 \\ -5 & -2 \\ 1 & -3 \\ 2 & -4 \end{bmatrix}, \begin{bmatrix} 3 & -5 \\ 4 & 1 \\ 5 & 2 \\ 6 & 3 \end{bmatrix}, \begin{bmatrix} 7 & 4 \\ 8 & 5 \\ 9 & 6 \\ 10 & 7 \end{bmatrix}, \ldots$$

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 7-313. 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,



the block outputs the following sequence of frames at the start of the simulation.

$$\begin{bmatrix} -1 & -2 \\ -3 & -4 \\ -5 & -6 \\ -7 & -8 \end{bmatrix}, \begin{bmatrix} -9 & -10 \\ 1 & 1 \\ 2 & 2 \\ 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 \\ 5 & 5 \\ 6 & 6 \\ 7 & 7 \end{bmatrix}, \ldots$$

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.

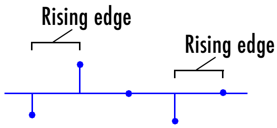
Resetting the Delay

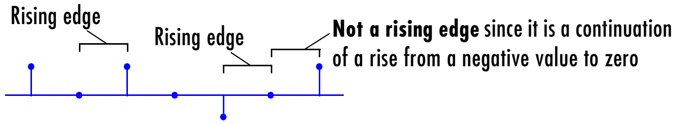
The block resets the delay whenever it detects a reset event at the optional `Rst` port. The reset signal rate must be a positive integer multiple of the rate of the data signal input.

The reset event is specified by the **Reset port** parameter, and can be one of the following:

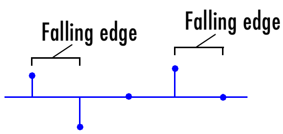
- **None** disables the `Rst` port.
- **Rising edge** — Triggers a reset operation when the `Rst` input does one of the following:
  - Rises from a negative value to a positive value or zero

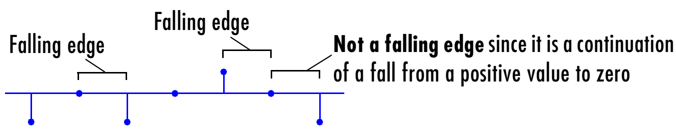
- Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)





- **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)





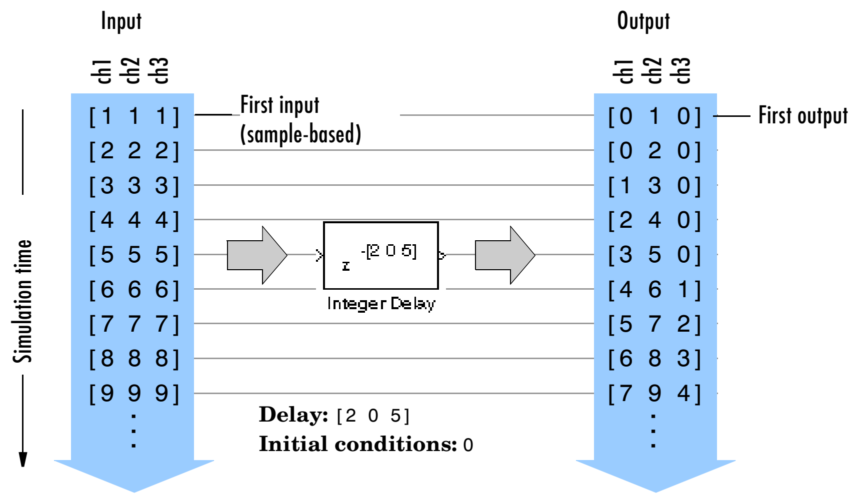
- **Either edge** — Triggers a reset operation when the `Rst` input is a **Rising edge** or **Falling edge** (as described above).
- **Non-zero sample** — Triggers a reset operation at each sample time that the `Rst` input is not zero.

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

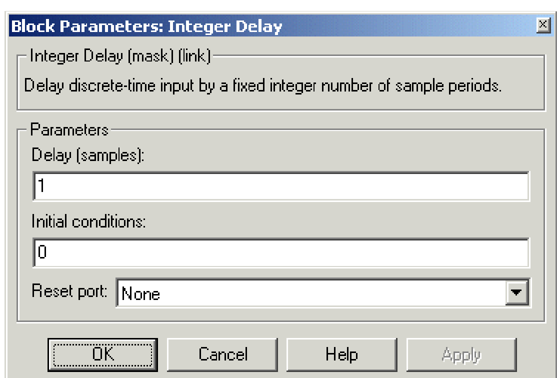
"Excess Algorithmic Delay (Tasking Latency)" on page 3-91 and the topic on the **Simulation Parameters** dialog box in the Simulink documentation.

Examples

The `dspafxr` demo illustrates an audio reverberation system built around the Integer Delay block.



Dialog Box



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 7-315.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed-point
- 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 to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

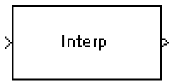
Unit Delay	Simulink
Variable Fractional Delay	DSP Blockset
Variable Integer Delay	DSP Blockset

Also see "Signal Operations" on page 7-16 for a list of all the blocks in the Signal Operations library.

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

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 **Parameter**.
- 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 7-320.

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:

- Set the **Source of interpolation points** parameter to **Input port**, which activates a block input port, In, for the interpolation points.
- Generate a signal of interpolation vectors with the *same frame status* and *same frame rate* as the input signal, and supply it to the input port for interpolation points. To learn about interpolation vectors, see "How the Block Applies Interpolation Vectors to Inputs" on page 7-320.

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:

- Apply the vector I_n to each channel of a matrix input, resulting in a matrix output.
- Apply 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).

Table 7-12: 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
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

Table 7-13: 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:

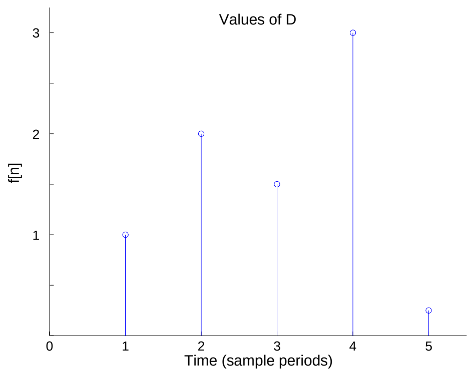
- `D = [11, 22, 33, 44]'`
- I_n `= [10, 2.6, -3]'`

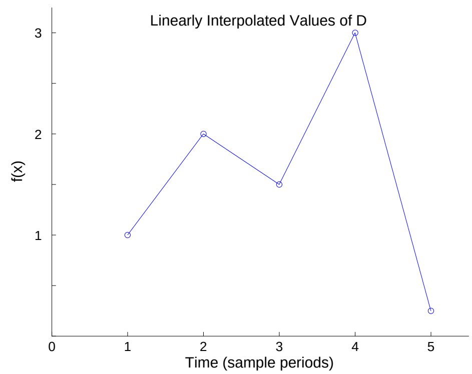
Since `D` has 4 samples, valid interpolation points range from 1 to 4. The block clips the interpolation point 10 to 4 and the point -3 to 1, resulting in the clipped interpolation vector $I_{nclipped}$ `= [4, 2.6, 1]'`.

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`.

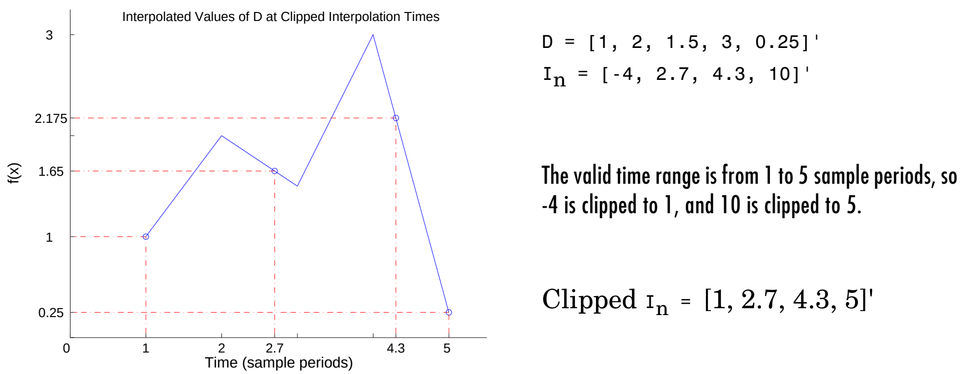




As illustrated below, if the block is in linear interpolation mode and is set to clip out-of-range interpolation points, where

- `D = [1, 2, 1.5, 3, 0.25]'`
- I_n `= [-4, 2.7, 4.3, 10]'`

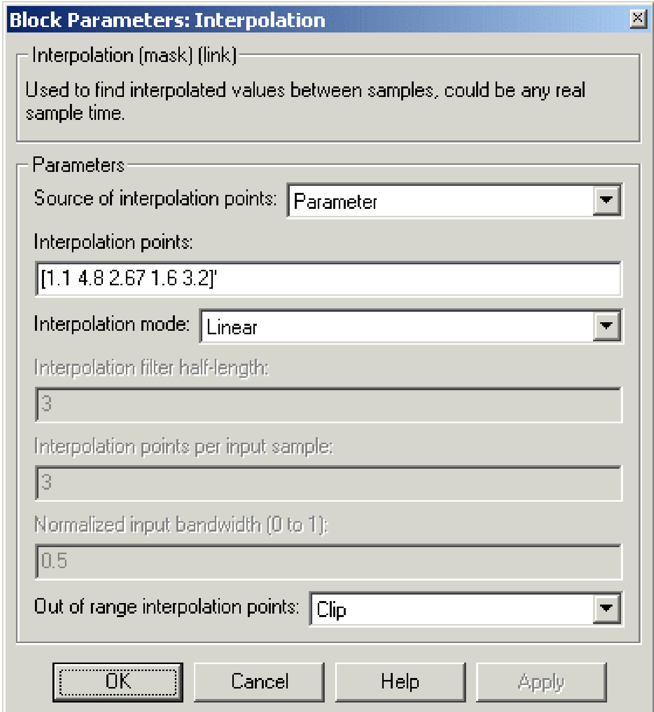
then the block clips the invalid interpolation points, and outputs the linearly interpolated values in a vector, `[1, 1.65, 2.175, 0.25]'`.



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 the topic on FIR Interpolation Mode in the Variable Fractional Delay block reference for more information.

Dialog Box



Source of interpolation points

Sets the location for specifying interpolation points (the points in time at which to interpolate the input): either in a dialog parameter (for static interpolation points) or a block input port (for time-varying interpolation points). For more information, see “Specifying Static Interpolation Points” on page 7-320 and “Specifying Time-Varying Interpolation Points” on page 7-320. Tunable.

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 7-320. Tunable.

Interpolation mode

Sets the block to interpolate by either linear or FIR interpolation. For more information, see “Linear Interpolation Mode” on page 7-323 and “FIR Interpolation Mode” on page 7-324.

Interpolator filter half-length

Half the length of the FIR interpolation filter. For more information, see “FIR Interpolation Mode” on page 7-324.

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. If 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 7-324. and “Linear Interpolation Mode” on page 7-323.

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 7-324.

Out of range interpolation points

If 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 7-322.

Supported Data Types

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

To learn how to convert to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

See “Signal Operations” on page 7-16 for a list of all the blocks in the Signal Operations library.

Kalman Adaptive Filter

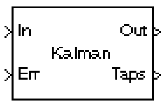
Purpose

Compute filter estimates for an input using the Kalman adaptive filter algorithm.

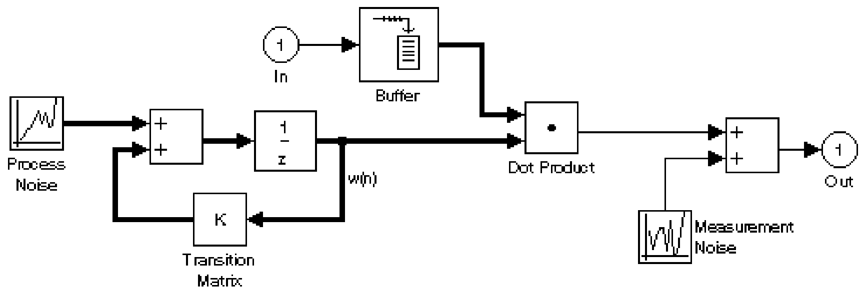
Library

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 dynamical system.



The Kalman filter assumes that there are no deterministic changes to the filter taps over time (i.e., 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

$$
\begin{aligned}
g(n) &= \frac{K(n-1)u(n)}{u^H(n)K(n-1)u(n)+Q_M} \\
y(n) &= u^H(n)\hat{w}(n) \\
e(n) &= d(n)-y(n) \\
\hat{w}(n+1) &= \hat{w}(n)+e(n)g(n) \\
K(n) &= K(n-1)-g(n)u^H(n)K(n-1)+Q_P
\end{aligned}
$$

The variables are as follows.

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. The signal at the `Out` port is a scalar, while the signal at the `Taps` port is a sample-based vector.

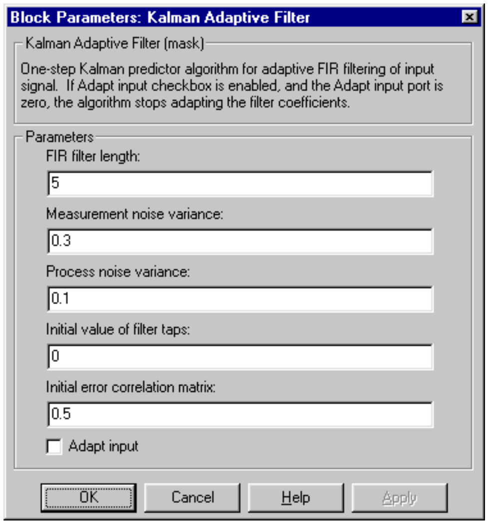
Block Ports	Corresponding Variables
`In`	u, the scalar input, which is internally buffered into the vector $u(n)$
`Out`	$y(n)$, the filtered scalar output
`Err`	$e(n)$, the scalar estimation error
`Taps`	$\hat{w}(n)$, the vector of filter-tap estimates

An optional `Adapt` input port is added when the **Adapt input** check box is selected 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.

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 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 to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

LMS Adaptive Filter	DSP Blockset
RLS Adaptive Filter	DSP Blockset

See “Adaptive Filters” on page 4-34 for related information. Also see a list of all blocks in the Adaptive Filters library.

LDL Factorization

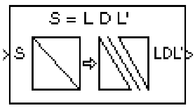
Purpose

Factor a square Hermitian positive definite matrix 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.

$$\begin{array}{|c|c|c|c|c|} \hline d_{11} & u_{12} & u_{13} & u_{14} & u_{15} \\ \hline l_{21} & d_{22} & u_{23} & u_{24} & u_{25} \\ \hline l_{31} & l_{32} & d_{33} & u_{34} & u_{35} \\ \hline l_{41} & l_{42} & l_{43} & d_{44} & u_{45} \\ \hline l_{51} & l_{52} & l_{53} & l_{54} & d_{55} \\ \hline \end{array} \qquad u_{ij} = l_{ji}^*$$

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.

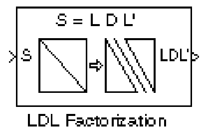
The following options are available:

- **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 ouput.
- **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 ouput.
- **Error** — Display an error dialog box and terminate the simulation.

Example

LDL decomposition of a 3-by-3 Hermitian positive definite matrix:

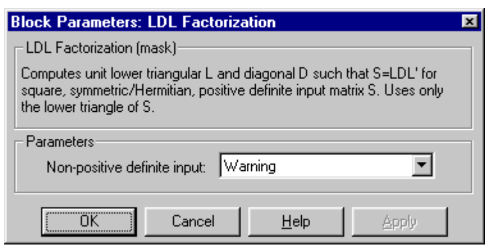
$$\begin{bmatrix} 9 & -1 & 2 \\ -1 & 8 & -5 \\ 2 & -5 & 7 \end{bmatrix}$$



$$\begin{bmatrix} 9.00 & -0.11 & 0.22 \\ -0.11 & 7.89 & -0.61 \\ 0.22 & -0.61 & 3.66 \end{bmatrix}$$

$$L = \begin{bmatrix} 1 & 0 & 0 \\ -0.11 & 1 & 0 \\ 0.22 & -0.61 & 1 \end{bmatrix} \qquad D = \begin{bmatrix} 9.00 & 0 & 0 \\ 0 & 7.89 & 0 \\ 0 & 0 & 3.66 \end{bmatrix} \qquad L' = \begin{bmatrix} 1 & -0.11 & 0.22 \\ 0 & 1 & -0.61 \\ 0 & 0 & 1 \end{bmatrix}$$

Dialog Box



Non-positive definite input

Response to non-positive definite matrix inputs. Tunable.

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 to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Cholesky Factorization	DSP Blockset
LDL Inverse	DSP Blockset
LDL Solver	DSP Blockset
LU Factorization	DSP Blockset
QR Factorization	DSP Blockset

See “Factoring Matrices” on page 6-8 for related information. Also see “Matrix Factorizations” on page 7-10 for a list of all the blocks in the Matrix Factorizations library.

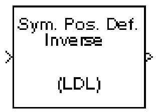
Purpose

Compute the inverse of a 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.

$$S^{-1} = (LDL^{*})^{-1}$$

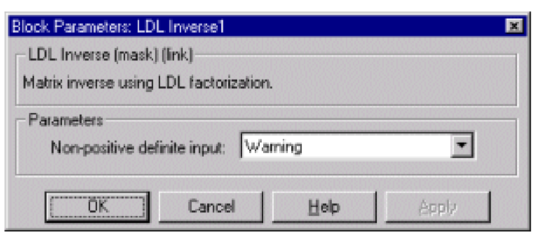
L is a lower triangular square matrix with unity diagonal elements, D is a diagonal matrix, and L^{*} is the Hermitian (complex conjugate) transpose of L. Only the diagonal and lower triangle of the input matrix are used, and any imaginary component of the diagonal entries is disregarded. The output 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.

Dialog Box



Non-positive definite input

Response to non-positive definite matrix inputs. Tunable.

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 to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

Cholesky Inverse	DSP Blockset
LDL Factorization	DSP Blockset
LDL Solver	DSP Blockset
LU Inverse	DSP Blockset
Pseudoinverse	DSP Blockset
`inv`	MATLAB

See "Inverting Matrices" on page 6-10 for related information. Also see "Matrix Inverses" on page 7-11 for a list of all the blocks in the Matrix Inverses library.

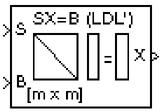
Purpose

Solve the equation SX=B for X when S is a 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-hand 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-hand 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.

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

$$Z = L^{*}X$$

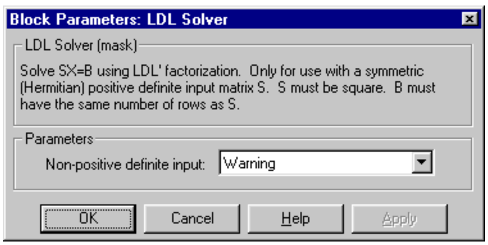
3 Solve one diagonal and two triangular systems.

$$LY = B$$

$$DZ = Y$$

$$L^{*}X = Z$$

Dialog Box



Non-positive definite input

Response to non-positive definite matrix inputs. Tunable.

Supported Data Types

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

To learn how to convert to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Autocorrelation LPC	DSP Blockset
Cholesky Solver	DSP Blockset
LDL Factorization	DSP Blockset
LDL Inverse	DSP Blockset
Levinson-Durbin	DSP Blockset
LU Solver	DSP Blockset
QR Solver	DSP Blockset

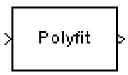
See “Solving Linear Systems” on page 6-7 for related information. Also see “Linear System Solvers” on page 7-9 for a list of all the blocks in the Linear System Solvers library.

Least Squares Polynomial Fit

Purpose Compute the coefficients of the polynomial that best fits the input data in a 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 n is specified by 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, $c_1, c_2, \ldots, c_{n+1}$, 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, $x_1, x_2, \ldots, x_M$, 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
```

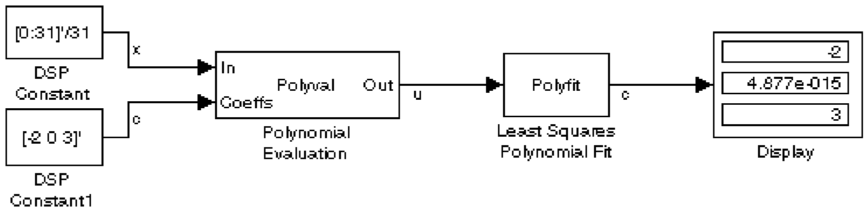
Inputs can be frame-based or sample-based. For convenience, a length-M 1-D vector input is treated as an M-by-1 matrix.

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, $c_1, c_2, \ldots, c_{n+1}$. The output is always sample-based.

Example In the model below, the Polynomial Evaluation block uses the second-order polynomial

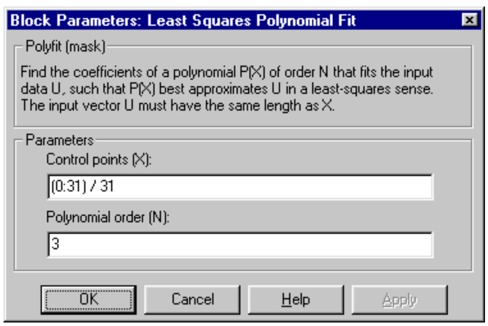
$$y = -2u^2 + 3$$

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. Tunable.

Supported Data Types

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

To learn how to convert to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Detrend	DSP Blockset
Polynomial Evaluation	DSP Blockset
Polynomial Stability Test	DSP Blockset
`polyfit`	MATLAB

Also see “Polynomial Functions” on page 7-12 for a list of all the blocks in the Polynomial Functions library.

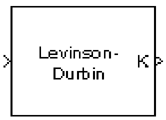
Purpose

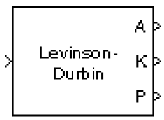
Solve a linear system of equations using Levinson-Durbin recursion.

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.

$$\begin{bmatrix} r(1) & r^*(2) & \cdots & r^*(n) \\ r(2) & r(1) & \cdots & r^*(n-1) \\ \vdots & \vdots & \ddots & \vdots \\ r(n) & r(n-1) & \cdots & r(1) \end{bmatrix} \begin{bmatrix} \mathrm{a}(2) \\ \mathrm{a}(3) \\ \vdots \\ \mathrm{a}(n+1) \end{bmatrix} = \begin{bmatrix} -r(2) \\ -r(3) \\ \vdots \\ -r(n+1) \end{bmatrix}$$

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 **K** is selected.) 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 **A and K** is selected.)

Both A and K are always 1-D vectors.

The prediction error power, P, (a scalar), is output when the **Output prediction error power (P)** check box is selected. 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 **If the value of lag 0 is zero, A=[1 zeros], K=[zeros], P=0** is selected (default), an input whose `r(1)` element is zero generates a zero-valued output. When this check box is *not* selected, an input with `r(1) = 0` generates `NaN`s 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 `NaN`s 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) ... 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.

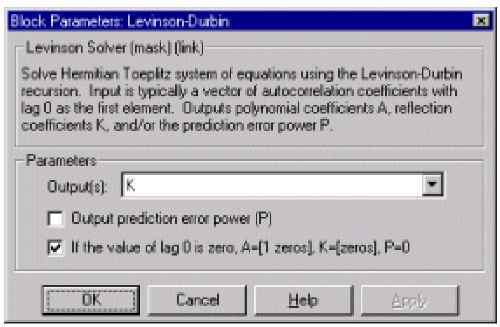
$$\min_{\{a_i\}} \quad E\left[\left|x_n - \sum_{i=1}^{N} a_i x_{n-i}\right|^2\right]$$

Again, the output at the block's `K` port contains the corresponding reflection coefficients, `[k(1) k(2) ... 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.

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



Output(s)

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)

When selected, the block outputs the prediction error at port `P`.

If the value of lag 0 is zero, A=[1 zeros], K=[zeros], P=0

When set, the block outputs a zero-vector for inputs having `r(1) = 0`. When cleared, the block outputs `NaN`s for these inputs.

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

To learn how to convert to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

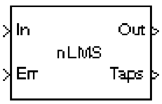
Cholesky Solver	DSP Blockset
LDL Solver	DSP Blockset
Autocorrelation LPC	DSP Blockset
LU Solver	DSP Blockset
QR Solver	DSP Blockset
Yule-Walker AR Estimator	DSP Blockset
Yule-Walker Method	DSP Blockset
`levinson`	Signal Processing Toolbox

See "Solving Linear Systems" on page 6-7 for related information. Also see "Linear System Solvers" on page 7-9 for a list of all the blocks in the Linear System Solvers library.

Purpose Compute filter estimates for an input using the LMS adaptive filter algorithm.

Library Filtering / Adaptive Filters

Description The LMS Adaptive Filter block implements an adaptive FIR filter using the stochastic gradient algorithm known as the normalized Least Mean-Square (LMS) algorithm.



$$\begin{aligned} y(n) &= \hat{w}^H(n-1)u(n) \\ e(n) &= d(n) - y(n) \\ \hat{w}(n) &= \hat{w}(n-1) + \frac{u(n)}{a + u^H(n)u(n)}\mu e^*(n) \end{aligned}$$

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 the **Adapt input** check box is selected 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.

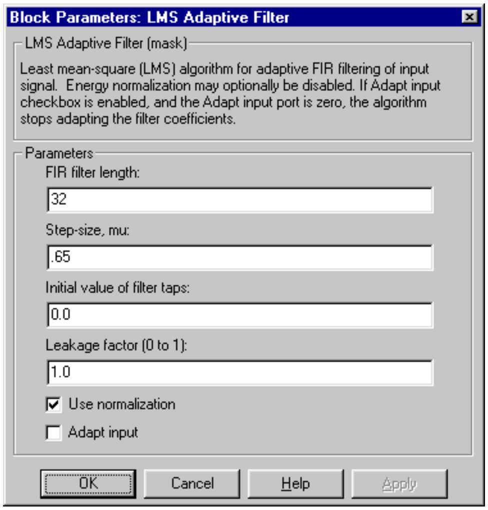
The **FIR filter length** parameter specifies the length of the filter that the LMS algorithm estimates. The **Step size** parameter corresponds to μ in the equations. Typically, for convergence in the mean square, 0<μ<2. The **Initial value of filter taps** specifies the initial value $\hat{w}(0)$ as a vector, or as a scalar to be repeated for all vector elements. The **Leakage factor** specifies the value of the leakage factor, $1 - \mu\alpha$, in the leaky LMS algorithm below. This parameter must be between 0 and 1.

$$\hat{w}(n+1) = (1 - \mu\alpha)\hat{\omega}(n) + \frac{u(n)}{u^H(n)u(n)}\mu e^*(n)$$

Examples

The `lmsdemo` demo illustrates a noise cancellation system built around the LMS Adaptive Filter block.

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 or clear normalization.

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 to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

Kalman Adaptive Filter	DSP Blockset
RLS Adaptive Filter	DSP Blockset

See "Adaptive Filters" on page 4-34 for related information. Also see a list of all blocks in the Adaptive Filters library.

LPC to LSF/LSP Conversion

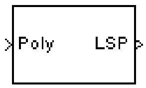
Purpose

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 polynomial 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, $1, a_1, a_2, \ldots, a_m$, must be the denominator of the transfer function of a stable all-pole filter with the form given in Equation 7-5. 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.

Sections of This Reference Page

- "Requirements for Getting Valid Outputs" on page 7-352 — Requirements that the input LPCs and the **Root finding coarse grid points** parameter value must meet to ensure valid block outputs
- "Setting Outputs to LSFs or LSPs" on page 7-353 — Descriptions of three possible output formats you must select with the **Output** parameter
- "Adjusting Output Computation Time and Accuracy with Root Finding Parameters" on page 7-353 — How to adjust the block's root finding time and accuracy with the **Root finding coarse grid points** and **Root finding bisection refinement** parameters
- "Valid Inputs and Corresponding Outputs" on page 7-354 — Valid input frame statuses, sizes, and dimensions, and those of the corresponding output
- "Handling and Recognizing Invalid Inputs and Outputs" on page 7-356 — How to set block parameters for handling invalid inputs and outputs
- "LSF and LSP Computation Method: Chebyshev Polynomial Method for Root Finding" on page 7-358 — Description and diagram of the block's root finding method

- "Root Finding Method Limitations: Failure to Find Roots" on page 7-361 — Description and diagram of how the block's root finding method can fail if parameters are not set properly
- "Dialog Box" on page 7-363 — A summary of the block parameters
- "Supported Data Types" on page 7-365 — Supported data types and a link to how to convert data types
- "See Also" on page 7-365 — Functions, blocks, and a paper related to the block

Requirements for Getting Valid Outputs

To get *valid outputs*, your inputs and the **Root finding coarse grid points** parameter value must meet these requirements:

- The input LPCs, $1, a_1, a_2, \ldots, a_m$, 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. If the input LPCs do not come from a transfer function of the following form, the block outputs are invalid.

$$H(z) = \frac{1}{1 + a_1 z^{-1} + a_2 z^{-2} + \ldots + a_m z^{-m}} \qquad \textbf{(7-5)}$$

- 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 7-358.) If you do not set **Root finding coarse grid points** to a high enough value relative to the number of LPCs, the block may not find all the LSPs or LSFs and yield invalid outputs as described in "Root Finding Method Limitations: Failure to Find Roots" on page 7-361.

To learn about recognizing invalid inputs and outputs and parameters for dealing with them, see "Handling and Recognizing Invalid Inputs and Outputs" on page 7-356.

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

$$\Delta LSF = \left|\operatorname{acos}(LSF_{act}) - \operatorname{acos}(LSF_{act} + 1/(n \cdot 2^k))\right|$$

Note If the value of the **Root finding coarse grid points** parameter is too small relative to the number of LPCs, the block may output *invalid data* as described in "Requirements for Getting Valid Outputs" on page 7-352. Also see "Handling and Recognizing Invalid Inputs and Outputs" on page 7-356.

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 Getting Valid Outputs" on page 7-352.
- 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 7-353):
  - LSFs — frequencies, w_k, where $0 < w_k < \pi$ and $w_k < w_{k+1}$
  - Normalized LSFs — $w_k / 2\pi$
  - LSPs — $\cos(w_k)$

Table 7-14: 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, $M > 0$ $\begin{bmatrix} 1 \ a_1 \ a_2 \ \dots \ a_m \end{bmatrix}$ *Frame-based row vectors are not valid inputs.*	Sample-based length-M row vector LSF in radians: $\begin{bmatrix} w_1 \ w_2 \ \dots \ w_m \end{bmatrix}$ LSF normalized: $\frac{1}{2\pi} \cdot \begin{bmatrix} w_1 \ w_2 \ \dots \ w_m \end{bmatrix}$ LSP: $\begin{bmatrix} \cos(w_1) \ \cos(w_2) \ \dots \ \cos(w_m) \end{bmatrix}$
Sample- or frame-based length-M+1 column vector, $M > 0$ $\begin{bmatrix} 1 \\ a_1 \\ a_2 \\ \vdots \\ a_m \end{bmatrix}$	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, $M > 0$ $(1, a_1, a_2, \dots, a_m)$	1-D length-M unoriented vector LSF in radians: $(w_1, w_2, \dots, w_m)$ LSF normalized: $\frac{1}{2\pi} \cdot (w_1, w_2, \dots, w_m)$ LSP: $(\cos(w_1), \cos(w_2), \dots, \cos(w_m))$

Handling and Recognizing Invalid Inputs and Outputs

The block outputs invalid data if your input LPCs and the value of the **Root finding coarse grid points** parameter do not meet the requirements described in "Requirements for Getting Valid Outputs" on page 7-352. 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 7-356
- "Parameters for Handling Invalid Inputs and Outputs" on page 7-357

What Invalid Outputs Look Like. Invalid outputs have the same dimensions, sizes, and frame statuses as valid outputs, which you can look up in Table 7-14, Input and Output Dimensions, Sizes, and Frame Statuses, on page 7-355. 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	Example Invalid Outputs
LSF in radians (0 pi)	π	$\begin{bmatrix} w_1 & w_2 & w_3 & \pi & \pi & \pi & \pi & \pi \end{bmatrix}$
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:

- **Output validity of current output (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* if the block fails to find all the LSF or LSP values or if the input LPCs are unstable (for details, see "Requirements for Getting Valid Outputs" on page 7-352). See the previous section to learn how to recognize invalid outputs.
- **If current output is invalid, overwrite with previous output** — Setting this parameter causes the block to overwrite invalid outputs with the *previous* output. If you set this parameter you also need to consider these parameters:
  a **When first output is invalid, overwrite with user-defined values** — If 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 *setting* this parameter 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.
  b **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 set the above parameter, **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 Table 7-14, Input and Output Dimensions, Sizes, and Frame Statuses, on page 7-355.
- **If first coefficient of input not 1** — The block output is invalid if 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.

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 "Reference" on page 7-365.

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 arccosine 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:
   - 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.
   - The block finds and outputs all M roots (given a length-M+1 LPC input)
   - The block fails to find all M roots and yields invalid outputs as described in "Handling and Recognizing Invalid Inputs and Outputs" on page 7-356.

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 b, Stop or Continue Root Finding Refinement.

   b **Stop or Continue Root Finding Refinement** — If the block has repeated Step a 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 a using the half-interval.

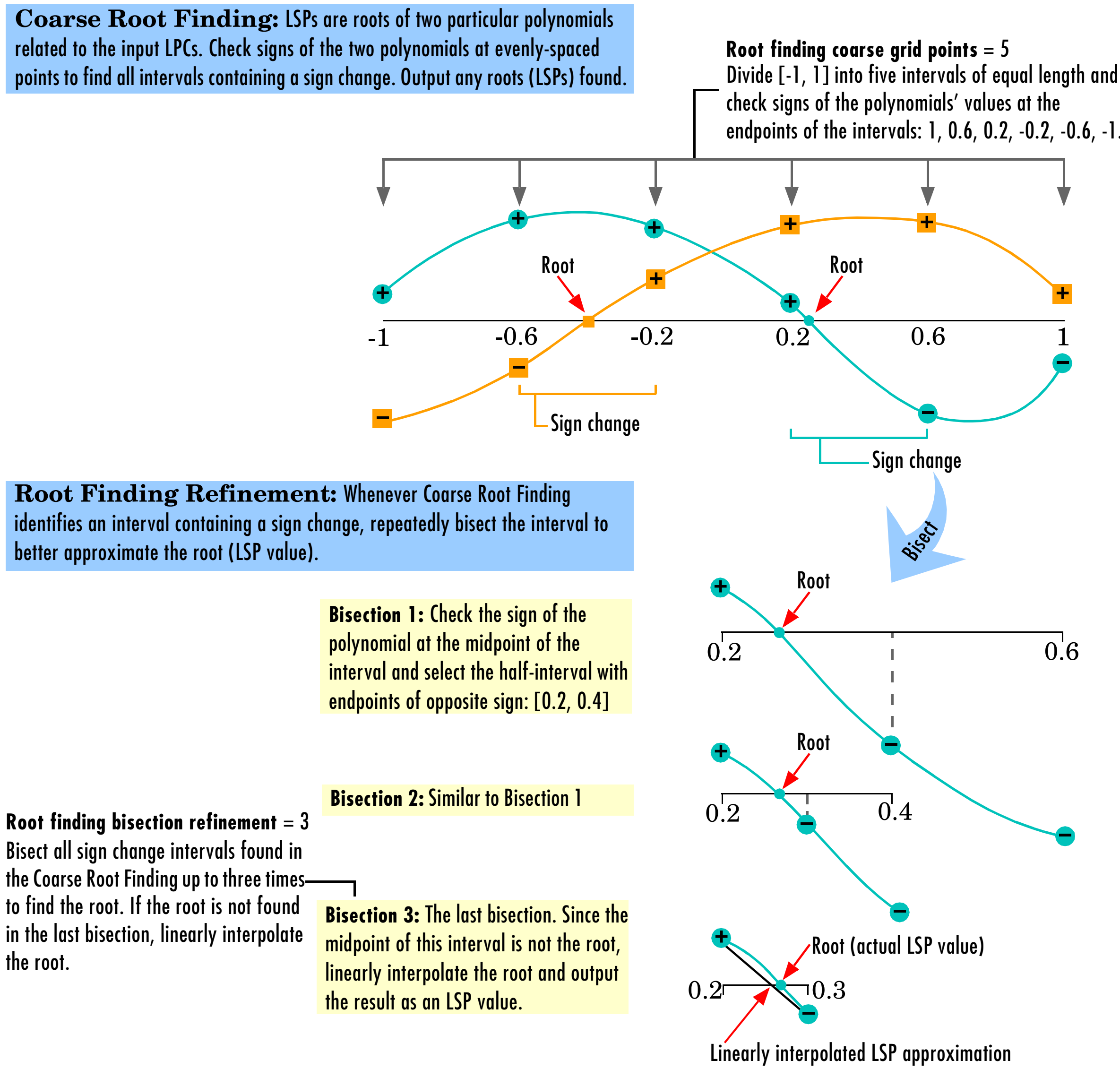


Figure 7-13: Coarse Root Finding and Root Finding Refinement

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 7-356).

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 may be too coarse to identify roots that are very close to each other, as illustrated in Figure 7-14, Fixing a Failed Root Finding, on page 7-362.

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.

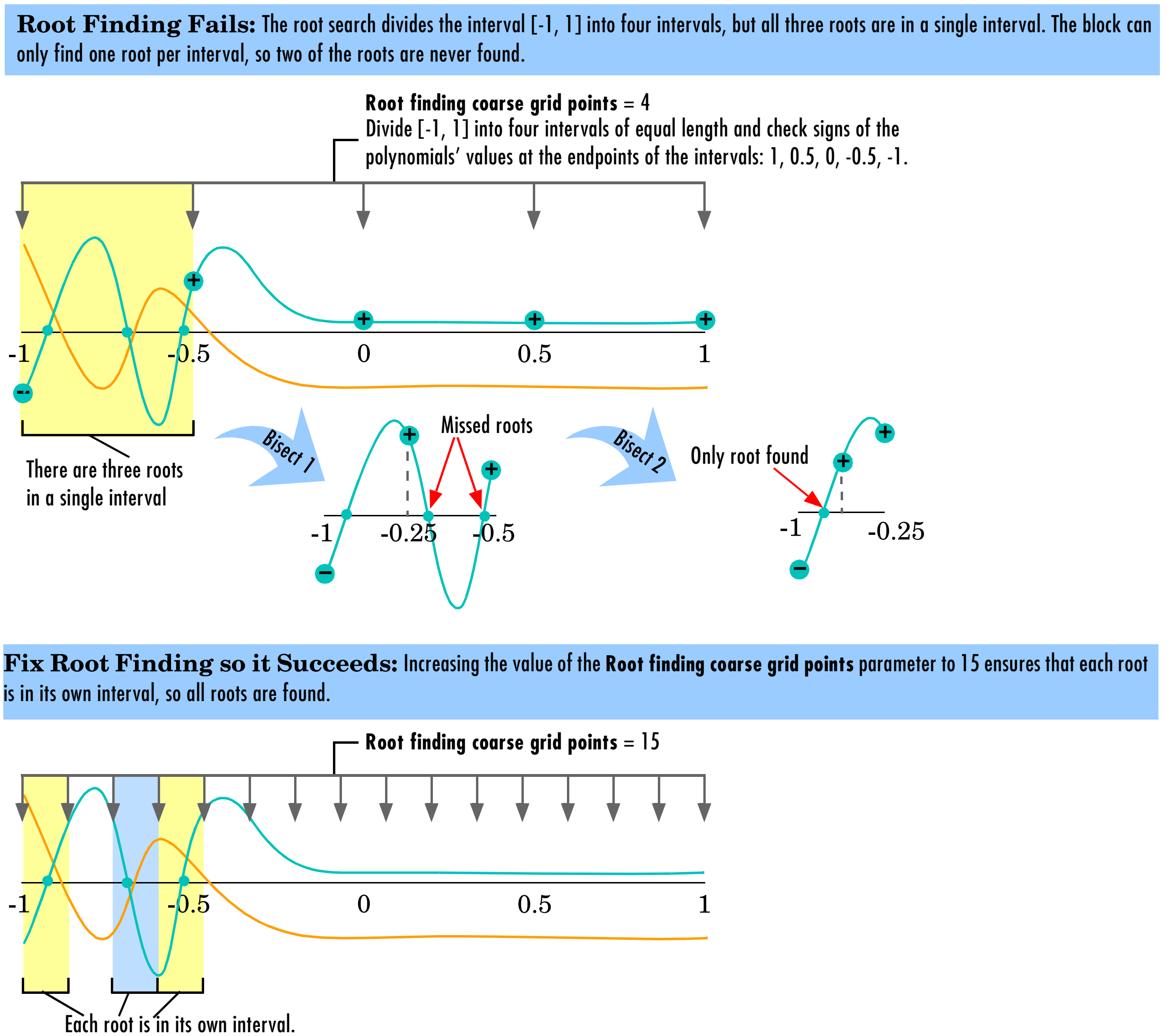
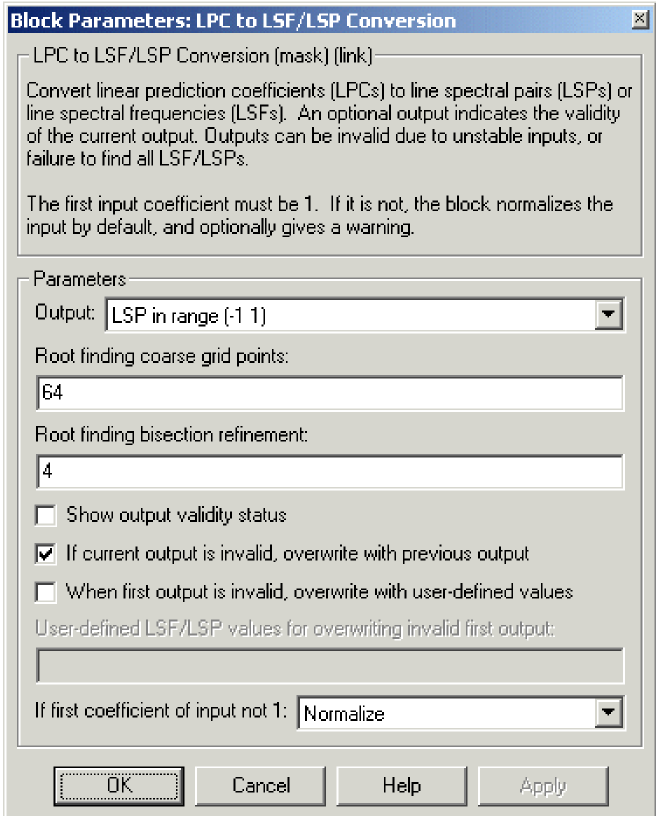


Figure 7-14: Fixing a Failed Root Finding

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 7-353 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 may be invalid as described in “Requirements for Getting Valid Outputs” on page 7-352. 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 7-358. Also see "Adjusting Output Computation Time and Accuracy with Root Finding Parameters" on page 7-353. 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 7-358. Also see "Adjusting Output Computation Time and Accuracy with Root Finding Parameters" on page 7-353. Tunable.

Output the validity of current output (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. For more information, see "Handling and Recognizing Invalid Inputs and Outputs" on page 7-356.

If current output is invalid, overwrite with previous output

Setting this parameter 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 7-357.

When first output is invalid, overwrite with user-defined values

If 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 *setting* this parameter). 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 7-357.

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 set the parameter **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 Table 7-14, Input and Output Dimensions, Sizes, and Frame Statuses, on page 7-355.

If first coefficient of input 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 if 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 7-357.

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, **Output the validity of current output (1=valid, 0=invalid)**

To learn how to convert to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

Reference

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

LSF/LSP to LPC Conversion	DSP Blockset
`poly2lsf`	Signal Processing Toolbox

Also see "Linear Prediction" on page 7-5 for a list of all the blocks in the Linear Prediction library.

LSF/LSP to LPC Conversion

Purpose

Convert line spectral frequencies (LSFs) or line spectral pairs (LSPs) to linear prediction coefficients (LPCs)

Library

Estimation / Linear Prediction

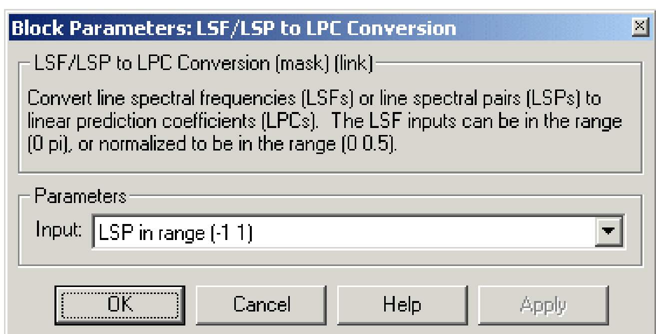
Description

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 radians (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



Output

Specifies whether to convert **LSP in range (-1 1)**, **LSF in radians (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 to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

Reference

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	DSP Blockset
`lsf2poly`	Signal Processing Toolbox

Also see "Linear Prediction" on page 7-5 for a list of all the blocks in the Linear Prediction library.

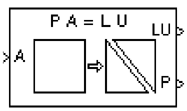
Purpose

Factor a square matrix into lower and upper triangular components.

Library

Math Functions / Matrices and Linear Algebra / Matrix Factorizations

Description



The LU Factorization block factors a row permutation 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 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
```

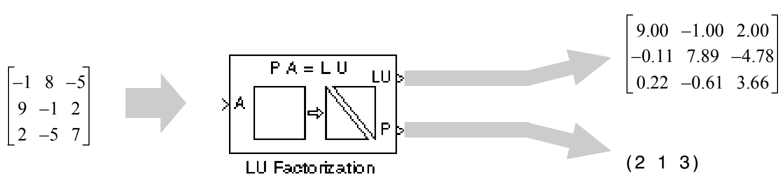
The output at the `LU` port is a composite matrix with lower subtriangle elements from L and upper triangle elements from U, and is always sample-based.

Example

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.

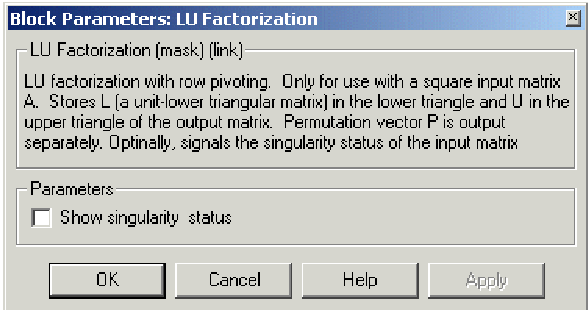
$$A = \begin{bmatrix} -1 & 8 & -5 \\ 9 & -1 & 2 \\ 2 & -5 & 7 \end{bmatrix} \qquad P = (2\ 1\ 3) \qquad A_p = \begin{bmatrix} 9 & -1 & 2 \\ -1 & 8 & -5 \\ 2 & -5 & 7 \end{bmatrix}$$

The `LU` output is a composite matrix whose lower subtriangle forms L and whose upper triangle forms U.



$$L = \begin{bmatrix} 1 & 0 & 0 \\ -0.11 & 1 & 0 \\ 0.22 & -0.61 & 1 \end{bmatrix} \qquad U = \begin{bmatrix} 9.00 & -1.00 & 2.00 \\ 0 & 7.89 & -4.78 \\ 0 & 0 & 3.66 \end{bmatrix}$$

Dialog Box



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.

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 singularity status**.

To learn how to convert to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

Autocorrelation LPC	DSP Blockset
Cholesky Factorization	DSP Blockset
LDL Factorization	DSP Blockset
LU Inverse	DSP Blockset
LU Solver	DSP Blockset
Permute Matrix	DSP Blockset
QR Factorization	DSP Blockset
`lu`	MATLAB

See “Factoring Matrices” on page 6-8 for related information. Also see “Matrix Factorizations” on page 7-10 for a list of all the blocks in the Matrix Factorizations library.

Purpose

Compute the inverse of a square matrix using LU factorization.

Library

Math Functions / Matrices and Linear Algebra / Matrix Inverses

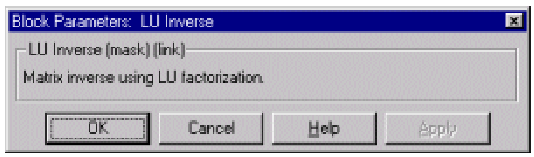
Description

The LU Inverse block computes the inverse of the square input matrix A by factoring and inverting row-pivoted variant A_p.

$$A_p^{-1} = (LU)^{-1}$$

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



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 to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

Cholesky Inverse	DSP Blockset
LDL Inverse	DSP Blockset
LU Factorization	DSP Blockset
LU Solver	DSP Blockset
`inv`	MATLAB

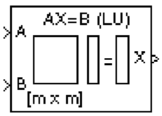
See "Inverting Matrices" on page 6-10 for related information. Also see "Matrix Inverses" on page 7-11 for a list of all the blocks in the Matrix Inverses library.

LU Solver

Purpose Solve the equation AX=B for X when A is a 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-hand 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-hand 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

$$A_pX = B_p$$

where B_p is the row-permuted variant of B, and the resulting equation

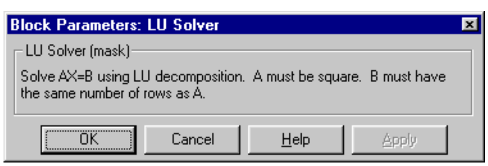
$$LUX = B_p$$

is solved for X by making the substitution Y = UX, and solving two triangular systems.

$$LY = B_p$$

$$UX = Y$$

Dialog Box



Supported Data Types

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

To learn how to convert to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Autocorrelation LPC	DSP Blockset
Cholesky Solver	DSP Blockset
LDL Solver	DSP Blockset
Levinson-Durbin	DSP Blockset
LU Factorization	DSP Blockset
LU Inverse	DSP Blockset
QR Solver	DSP Blockset

See “Solving Linear Systems” on page 6-7 for related information. Also see “Linear System Solvers” on page 7-9 for a list of all the blocks in the Linear System Solvers library.

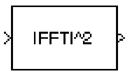
Purpose

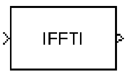
Compute a nonparametric estimate of the spectrum using the periodogram method.

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 **Inherit FFT length from input dimensions** is selected, N_{fft} is specified by the frame size of the input, which must be a power of 2. When **Inherit FFT length from input dimensions** is *not* selected, 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.

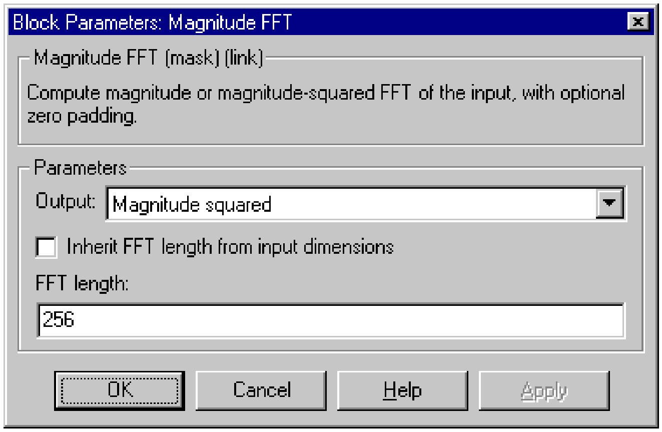
Each column of the output matrix contains the estimate of the corresponding input column's power spectral density at N_{fft} equally spaced frequency points in the range $[0,F_s)$, where F_s is the signal's sample frequency. The output is always sample-based.

The block does not accept sample-based 1-by-N row vector inputs.

Example

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. Tunable.

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 size

The number of data points on which to perform the FFT, N_{fft}. If N_{fft} exceeds the input frame size, the frame is zero-padded as needed. This parameter is enabled when **Inherit FFT length from input dimensions** is not selected.

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 to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Burg Method	DSP Blockset
Short-Time FFT	DSP Blockset
Spectrum Scope	DSP Blockset
Yule-Walker Method	DSP Blockset
`pwelch`	Signal Processing Toolbox

See "Power Spectrum Estimation" on page 6-6 for related information. Also see a list of all blocks in the Power Spectrum Estimation library and the Transforms library.

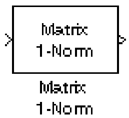
Purpose

Compute the 1-norm of a matrix.

Library

Math Functions / Matrices and Linear Algebra / Matrix Operations

Description

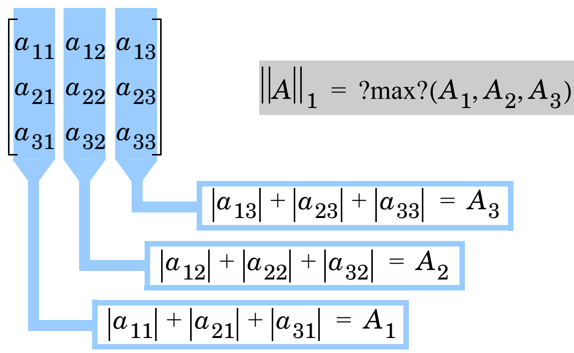


The Matrix 1-Norm block computes the 1-norm, or maximum column-sum, of an M-by-N input matrix, A.

$$y = \|A\|_1 = ?\text{max}?_{1 \le j \le N} \sum_{i=1}^{M} |a_{ij}|$$

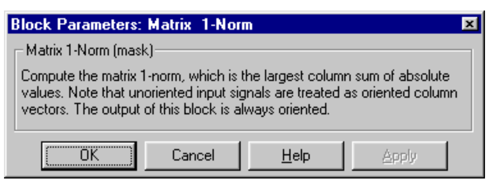
This is equivalent to

```
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.

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 to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

Normalization	DSP Blockset
Reciprocal Condition	DSP Blockset
`norm`	MATLAB

Also see "Matrix Operations" on page 7-11 for a list of all the blocks in the Matrix Operations library.

Purpose

Multiply input matrices.

Library

Math Functions / Matrices and Linear Algebra / Matrix Operations

Description

The Matrix Multiply block multiplies n input matrices, A, B, C, ..., U_n, in the forward direction, where n is specified by the **Number of input ports** parameter and U_n is the input at the nth port.

```
Y = ((((A*B)*C)*D) ... Un)      % Equivalent MATLAB code
```

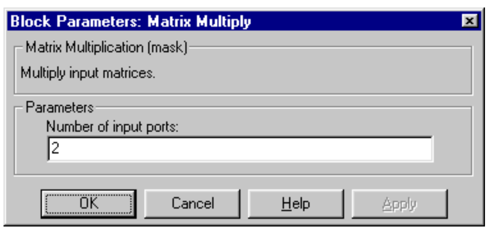
All inputs must have sizes compatible for matrix multiplication; that is, `size(A,2)` = `size(B,1)`, `size(B,2)` = `size(C,1)`, and so on. Inputs can be real, complex, sample-based, or frame-based in any combination, but *all* inputs must have the same precision, single or double. A length-M 1-D vector input at any port is treated as an M-by-1 matrix.

The size of sample-based output `Y` is `[size(A,1) size(Un,2)]`. That is, `Y` is M_A-by-N_{Un}.

Algorithm

The Matrix Multiply block is optimized to use at most two temporary variables for storage of intermediate results.

Dialog Box



Number of input ports
The number of inputs to the block.

Supported Data Types

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

To learn how to convert to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Dot Product	Simulink
Matrix Product	DSP Blockset
Matrix Scaling	DSP Blockset
Product	Simulink

Also see “Matrix Operations” on page 7-11 for a list of all the blocks in the Matrix Operations library.

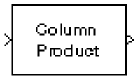
Purpose

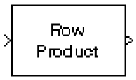
Multiply the elements of a matrix along rows or columns.

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.

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \\ u_{31} & u_{32} & u_{33} \end{bmatrix} \Rightarrow \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} (u_{11}u_{12}u_{13}) \\ (u_{21}u_{22}u_{23}) \\ (u_{31}u_{32}u_{33}) \end{bmatrix}$$

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.

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \\ u_{31} & u_{32} & u_{33} \end{bmatrix}$$

$$\Downarrow$$

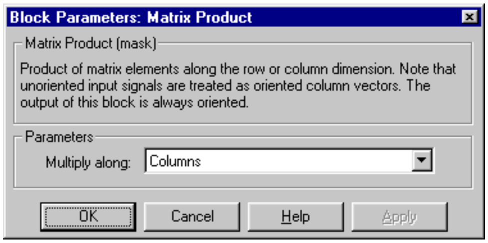
$$\begin{bmatrix} y_1 & y_2 & y_3 \end{bmatrix} = \begin{bmatrix} (u_{11}u_{21}u_{31}) & (u_{12}u_{22}u_{32}) & (u_{13}u_{23}u_{33}) \end{bmatrix}$$

This is equivalent to

```
y = prod(u)        % Equivalent MATLAB code
```

The output has the same frame status as the input.

Dialog Box



Multiply along

The dimension of the matrix along which to multiply, row or column.

Supported Data Types

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

To learn how to convert to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Matrix Multiply	DSP Blockset
Matrix Square	DSP Blockset
Matrix Sum	DSP Blockset
`prod`	MATLAB

Also see “Matrix Operations” on page 7-11 for a list of all the blocks in the Matrix Operations library.

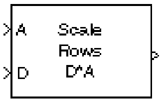
Purpose

Scale the rows or columns of a matrix by a 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)**, 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`.

$$\begin{bmatrix} d_1 \\ d_2 \\ d_3 \end{bmatrix} \begin{matrix} \times \\ \times \\ \times \end{matrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \Rightarrow \begin{bmatrix} d_1 a_{11} & d_1 a_{12} & d_1 a_{13} \\ d_2 a_{21} & d_2 a_{22} & d_2 a_{23} \\ d_3 a_{31} & d_3 a_{32} & d_3 a_{33} \end{bmatrix}$$

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)**, 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`.

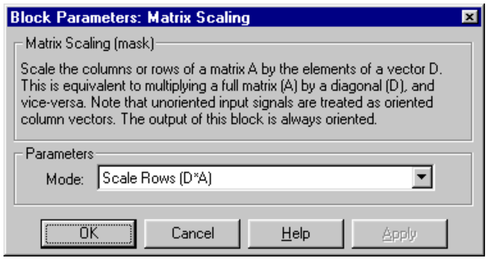
$$\begin{matrix} \begin{bmatrix} d_1 & d_2 & d_3 \end{bmatrix} \\ \times \quad \times \quad \times \\ \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \end{matrix} \Rightarrow \begin{bmatrix} d_1 a_{11} & d_2 a_{12} & d_3 a_{13} \\ d_1 a_{21} & d_2 a_{22} & d_3 a_{23} \\ d_1 a_{31} & d_2 a_{32} & d_3 a_{33} \end{bmatrix}$$

This is equivalent to postmultiplying `A` by a diagonal matrix with diagonal `D`.

```
y = A*diag(D)      % Equivalent MATLAB code
```

The output is the same size as the input matrix, `A`. If both inputs are sample-based, the output is sample-based; otherwise, the output is frame-based.

Dialog Box



Mode

The mode of operation, row scaling or column scaling. Tunable.

Supported Data Types

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

To learn how to convert to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Matrix Multiply	DSP Blockset
Matrix Product	DSP Blockset
Matrix Sum	DSP Blockset

Also see “Matrix Operations” on page 7-11 for a list of all the blocks in the Matrix Operations library.

Purpose

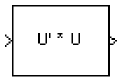
Compute the square of the input matrix.

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.




```
y = u' * u        % Equivalent MATLAB code
```

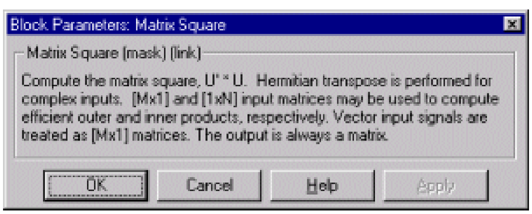
A length-M 1-D vector input is treated as an M-by-1 matrix. For both sample-based and frame-based inputs, output `y` is sample-based with dimension N-by-N.

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



Supported Data Types

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

To learn how to convert to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

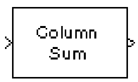
Matrix Multiply	DSP Blockset
Matrix Product	DSP Blockset
Matrix Sum	DSP Blockset
Transpose	DSP Blockset

Also see "Matrix Operations" on page 7-11 for a list of all the blocks in the Matrix Operations library.

Purpose Sum the elements of a matrix along rows or columns.

Library Math Functions / Matrices and Linear Algebra / Matrix Operations

Description 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.

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \\ u_{31} & u_{32} & u_{33} \end{bmatrix} \Rightarrow \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} u_{11} + u_{12} + u_{13} \\ u_{21} + u_{22} + u_{23} \\ u_{31} + u_{32} + u_{33} \end{bmatrix}$$

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.

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \\ u_{31} & u_{32} & u_{33} \end{bmatrix}$$

$$\Downarrow$$

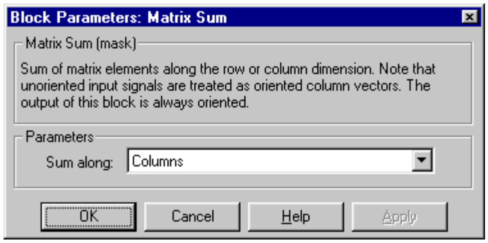
$$\begin{bmatrix} y_1 & y_2 & y_3 \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{3} u_{i1} & \sum_{i=1}^{3} u_{i2} & \sum_{i=1}^{3} u_{i3} \end{bmatrix}$$

This is equivalent to

```
y = sum(u)          % Equivalent MATLAB code
```

The output has the same frame status as the input.

Dialog Box



Sum along

The dimension of the matrix to sum along, row or column.

Supported Data Types

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

To learn how to convert to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Matrix Product	DSP Blockset
Matrix Multiply	DSP Blockset
`sum`	MATLAB

Also see “Matrix Operations” on page 7-11 for a list of all the blocks in the Matrix Operations library.

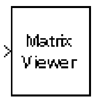
Purpose

Display a matrix as a color image.

Library

DSP 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. (A length-M 1-D vector input is treated as an M-by-1 matrix.)

Image Properties

Click on the **Image properties** check box to expose the image property parameters, which control the colormap and display.

The mapping of matrix element values to colors is specified by the **Colormap matrix**, **Minimum input**, and **Maximum input** 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 `ColorSpec` 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, if 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 is used to represent the value specified by the **Minimum input** parameter, and the color in the last row is used to represent the value specified by the **Maximum input** parameter. Values between the minimum and maximum are quantized and mapped to the intermediate rows of the colormap matrix.

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

Click on the **Axis properties** check box to expose 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 **Upper left corner** is specified, 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 **Lower left corner** is specified, 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. Menus and axis titles are not displayed. This option can also be selected from the right-click pop-up menu in the figure window.

When **Axis zoom** is cleared, the axis labels and titles are displayed in a gray border surrounding the image axes, and the window's menus (including **Axes**) and toolbar are visible. The Plot Editor tools allow you to annotate and customize the image display. Select **Help Plot Editor** from the figure's **Help** menu for more information about using these tools. For information on printing or saving the image, or on the other options found in the generic figure menus (**File**, **Edit**, **Window**, **Help**), see the MATLAB documentation.

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).

In addition to the standard MATLAB figure window menus (**File**, **Edit**, **Window**, **Help**), the Matrix Viewer window has an **Axes** menu containing the following items:

- **Refresh** erases all data on the scope display, except for the most recent image.
- **Autoscale** recomputes the **Minimum input** and **Maximum input** parameter values to best 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** and **Maximum input** parameters, where you can make further adjustments to them manually.
- **Axis zoom**, when selected, causes the image to completely fill the containing figure window. Menus and 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, and the window's menus (including **Axes**) and toolbar are visible. This option can also be set in the **Axis properties** panel 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** field 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, simply drag the window to the desired location, resize it as needed, and select **Save Position**. Note that the parameter dialog box must be closed when you select **Save Position** in order for the **Figure position** parameter to be updated.

Many of these options can also be accessed by right-clicking with the mouse anywhere on the displayed image. The right-click menu is very helpful when the scope is in zoomed mode and the **Axes** menu is not visible.

Examples

See the demo `dspstfft.mdl` for an example of using the Matrix Viewer block to create a moving spectrogram (time-frequency plot) of a speech signal by updating just one column of the input matrix at each sample time.

Dialog Box

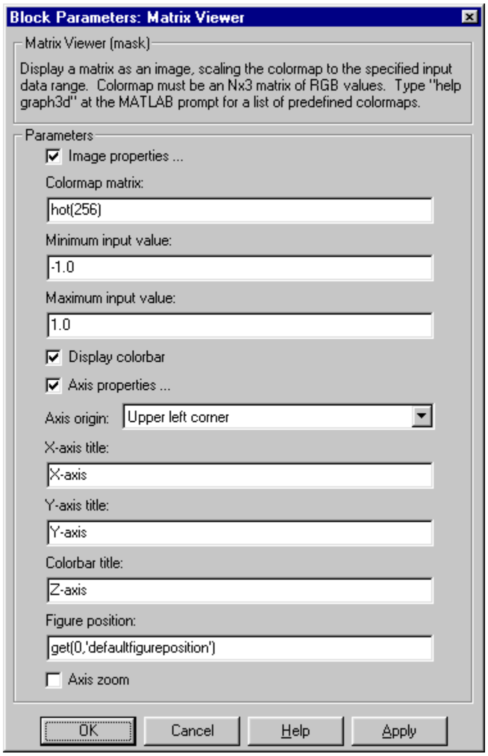


Image properties

Select to expose the image property parameters. Tunable.

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 `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. Select **Autoscale** from the right-click 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. Select **Autoscale** from the right-click 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 properties

Select to expose the axis property parameters. 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, if **Display colorbar** is currently selected. Tunable.

Figure position

A 4-element vector of the form `[left bottom 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

- 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 to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Spectrum Scope	DSP Blockset
Vector Scope	DSP Blockset
`colormap`	MATLAB
`ColorSpec`	MATLAB
`image`	MATLAB

Also see the following topics:

- “Viewing Signals” on page 3-80 — How to use this and other blocks to view signals
- “DSP Sinks” on page 7-3 — List of all blocks in the DSP Sinks library

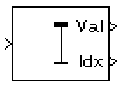
Purpose

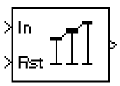
Find the maximum values in an input or sequence of inputs.

Library

Statistics

Description





The Maximum block identifies the value and 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**.

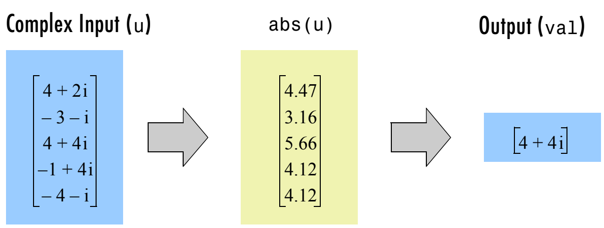
Value Mode

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*, `max(abs(u))`, as shown below.



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.

If a maximum value occurs more than once in a particular column of `u`, the computed index corresponds to the first occurrence. For example, if 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 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 jth column of all inputs since the last reset, up to and including element u_{ij} of the current input.

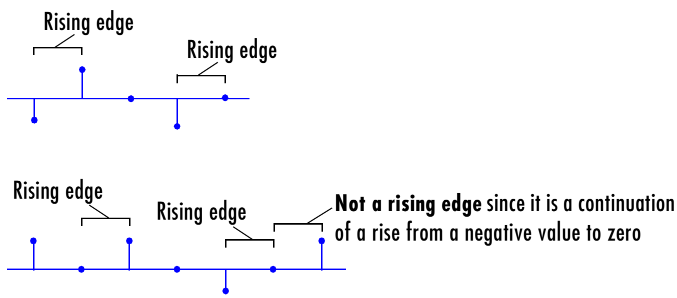
As in the other modes, length-M 1-D vector inputs and *sample-based* length-M row vector inputs are both treated as M-by-1 column vectors.

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.

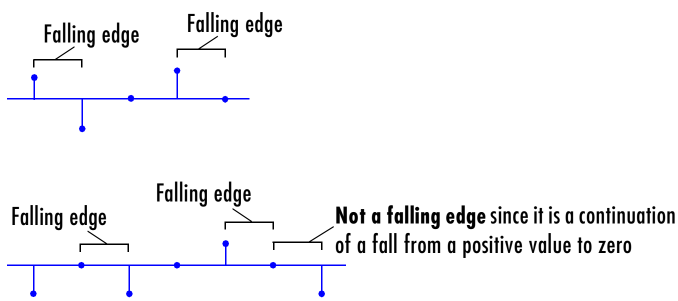
For sample-based inputs, a reset event causes the running maximum for each channel to be initialized to the value in the corresponding channel of the current input. For frame-based inputs, a reset event causes the running maximum for each channel to be initialized to the earliest value in each channel of the current input.

The reset event is specified by the **Reset port** menu, 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)

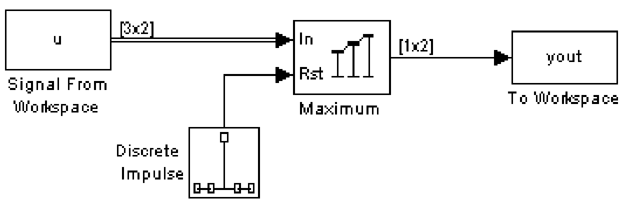


- **Either edge** — Triggers a reset operation when the `Rst` input is a **Rising edge** or **Falling edge** (as described above).
- **Non-zero sample** — Triggers a reset operation at each sample time that the `Rst` input is not zero.

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 3-91 and the topic on the **Simulation Parameters** dialog box in the Simulink documentation.

Example

The Maximum block in the model below 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

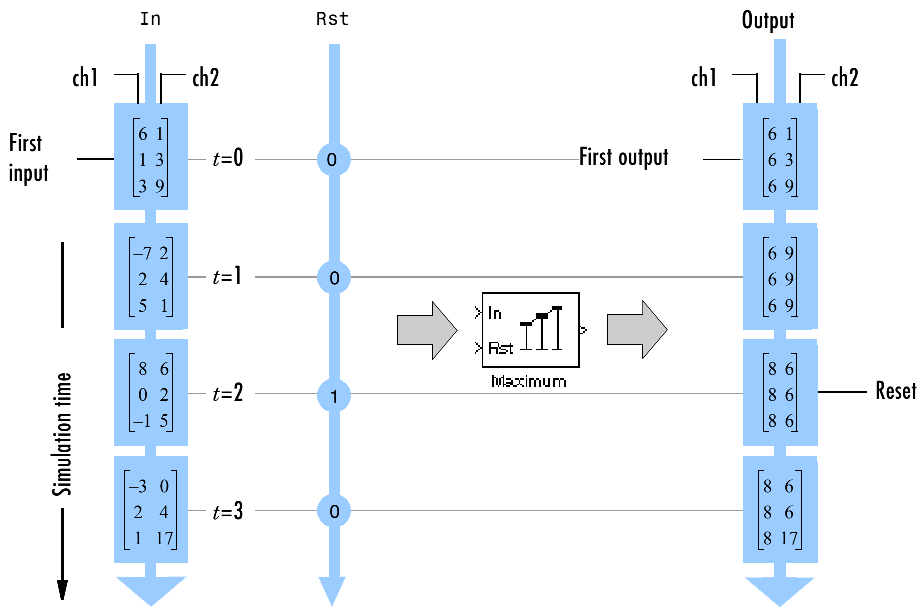
```
u = [6 1 3 -7 2 5 8 0 -1 -3 2 1;1 3 9 2 4 1 6 2 5 0 4 17]'
```

The Discrete Impulse block has the following settings:

- **Delay (samples)** = `2`
- **Sample time** = `1`

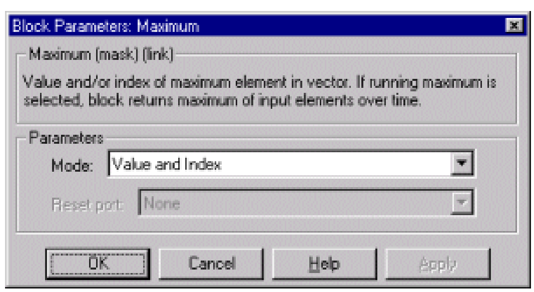
- **Samples per frame** = 1

The block's operation is shown in the figure below.



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

Dialog Box



Mode

The block's mode of operation: Output the maximum value of each input (**Value**), the index of the maximum value (**Index**), both the value and the index (**Value and index**), or track the maximum value of the input sequence over time (**Running**).

Reset port

Specifies the reset event detected at the `Rst` input port when **Running** is selected as 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 options for this parameter, see "Resetting the Running Maximum" on page 7-396.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- 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 to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

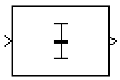
Mean	DSP Blockset
Minimum	DSP Blockset
MinMax	Simulink
`max`	MATLAB

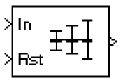
Also see "Statistics" on page 7-18 for a list of all the blocks in the Statistics library.

Purpose Find the 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.





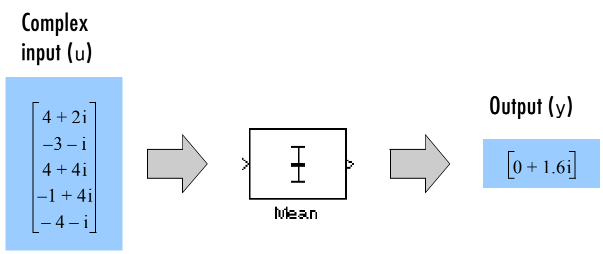
Basic Operation

When the **Running mean** check box is *not* selected, 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.



The frame status of the output is the same as that of the input.

Running Operation

When the **Running mean** check box is selected, 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.

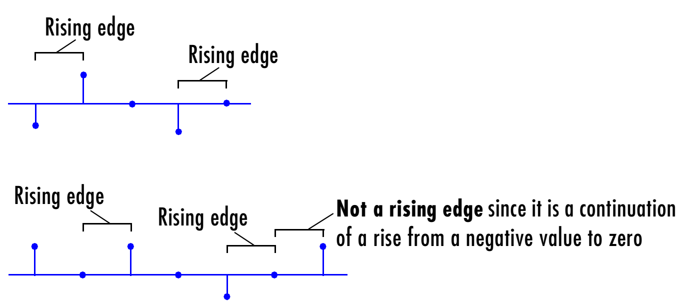
As in basic operation, length-M 1-D vector inputs and *sample-based* length-M row vector inputs are both treated as M-by-1 column vectors.

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.

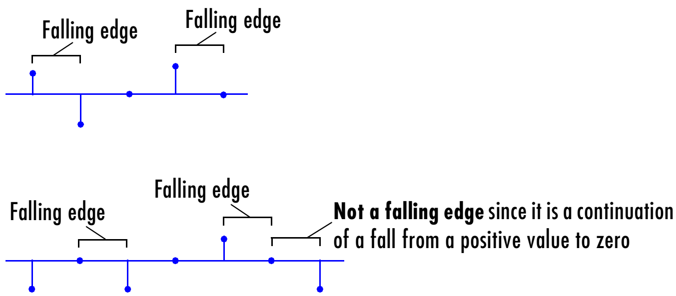
The reset event is specified by the **Reset port** parameter, and can be one of the following:

- **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)

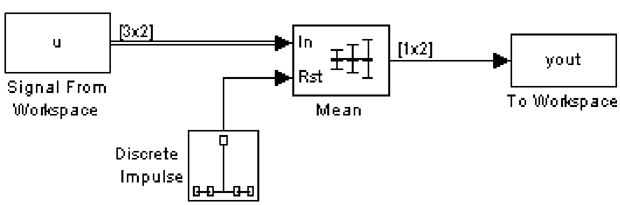


- **Either edge** — Triggers a reset operation when the `Rst` input is a **Rising edge** or **Falling edge** (as described above).
- **Non-zero sample** — Triggers a reset operation at each sample time that the `Rst` input is not zero.

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 3-91 and the topic on the **Simulation Parameters** dialog box in the Simulink documentation.

Example

The Mean block in the model below 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.



The Mean block has the following settings:

- **Running mean** = ☑
- **Reset port** = **Non-zero sample**

The Signal From Workspace block has the following settings:

- **Signal** = u
- **Sample time** = 1/3
- **Samples per frame** = 3

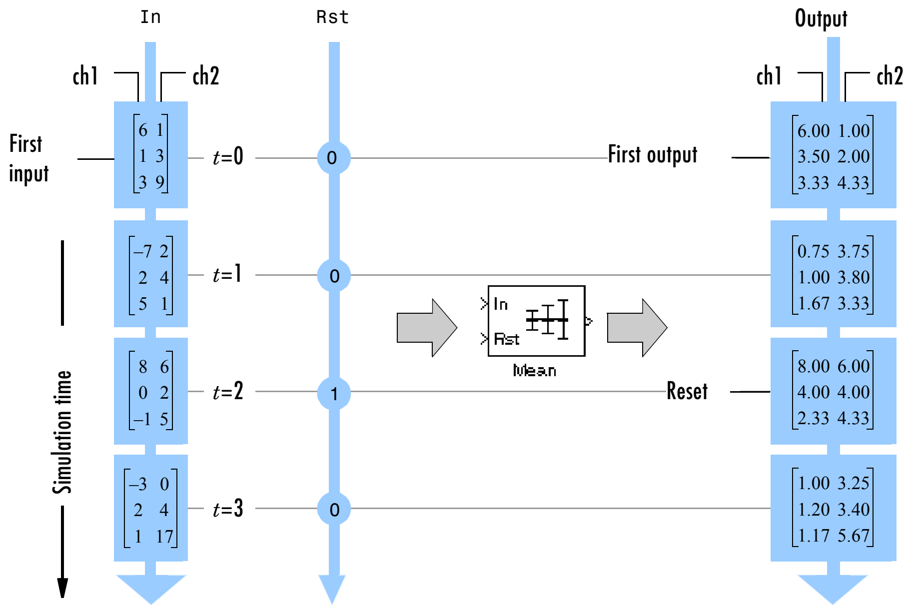
where

```
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:

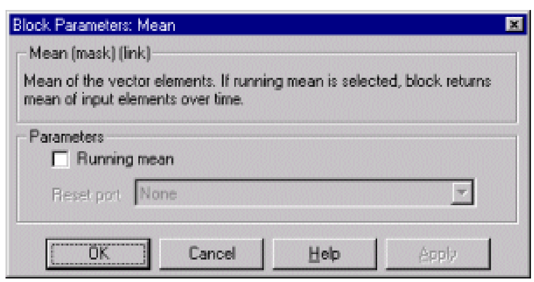
- **Delay (samples)** = 2
- **Sample time** = 1
- **Samples per frame** = 1

The block's operation is shown in the figure below.



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

Dialog Box



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 Mean” on page 7-402.

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 to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

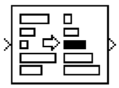
Maximum	DSP Blockset
Median	DSP Blockset
Minimum	DSP Blockset
Standard Deviation	DSP Blockset
`mean`	MATLAB

Also see “Statistics” on page 7-18 for a list of all the blocks in the Statistics library.

Purpose Find the 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.

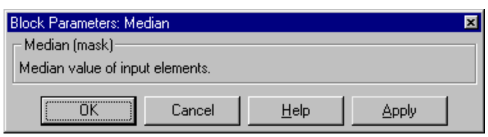
```
s = sort(u);
y = s((M+1)/2,:)
```

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, and the real and imaginary components are averaged independently (for even M).

Dialog Box



Supported Data Types

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

To learn how to convert to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Maximum	DSP Blockset
Mean	DSP Blockset
Minimum	DSP Blockset
Sort	DSP Blockset
Standard Deviation	DSP Blockset
Variance	DSP Blockset
`median`	MATLAB

Also see “Statistics” on page 7-18 for a list of all the blocks in the Statistics library.

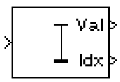
Purpose

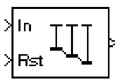
Find the minimum values in an input or sequence of inputs.

Library

Statistics

Description





The Minimum block identifies the value and 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**.

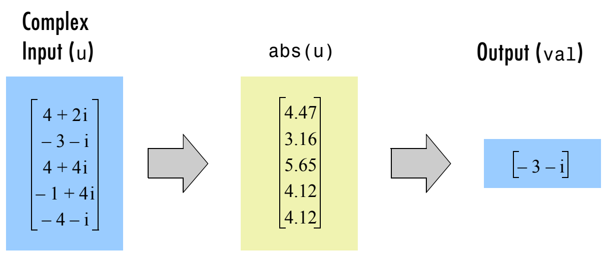
Value Mode

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*, `min(abs(u))`, as shown below.



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.

If a minimum value occurs more than once in a particular column of u, the computed index corresponds to the first occurrence. For example, if 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 jth column of all inputs since the last reset, up to and including element u_{ij} of the current input.

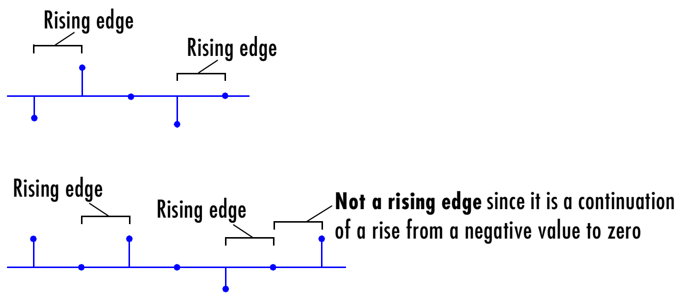
As in the other modes, length-M 1-D vector inputs and *sample-based* length-M row vector inputs are both treated as M-by-1 column vectors.

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.

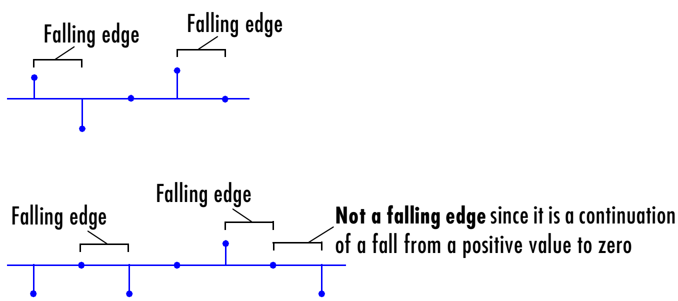
When the block is reset for sample-based inputs, the running minimum for each channel is initialized to the value in the corresponding channel of the current input. For frame-based inputs, the running minimum for each channel is initialized to the earliest value in each channel of the current input.

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)

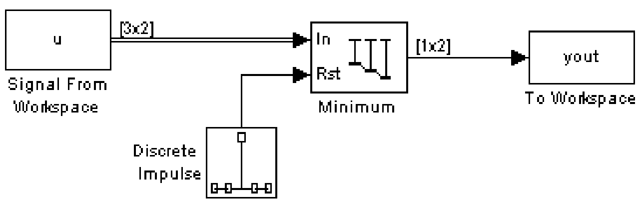


- **Either edge** — Triggers a reset operation when the `Rst` input is a **Rising edge** or **Falling edge** (as described above).
- **Non-zero sample** — Triggers a reset operation at each sample time that the `Rst` input is not zero.

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 3-91 and the topic on the **Simulation Parameters** dialog box in the Simulink documentation.

Example

The Minimum block in the model below 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**

The Signal From Workspace block has the following settings:

- **Signal** = `u`
- **Sample time** = `1/3`
- **Samples per frame** = `3`

where

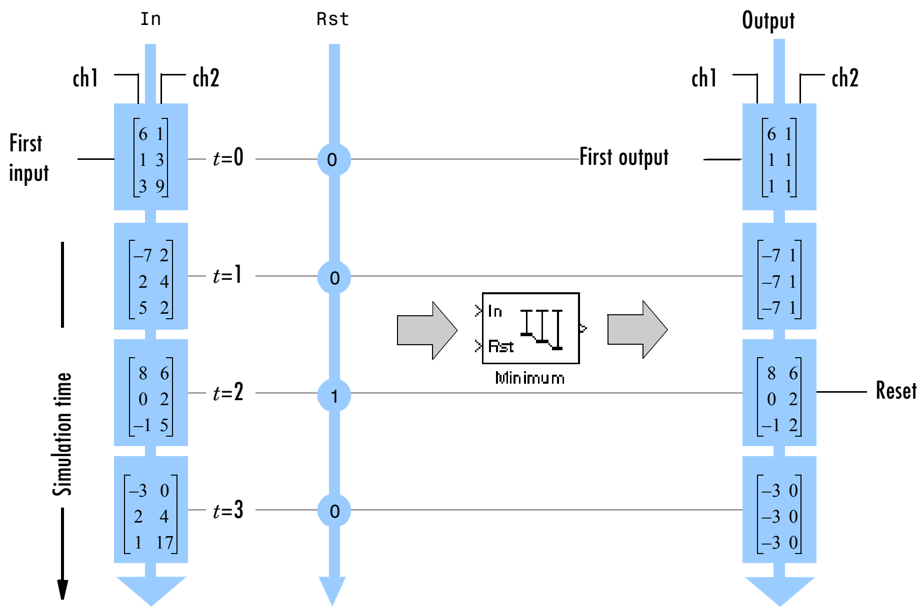
```
u = [6 1 3 -7 2 5 8 0 -1 -3 2 1;1 3 9 2 4 2 6 2 5 0 4 17]'
```

The Discrete Impulse block has the following settings:

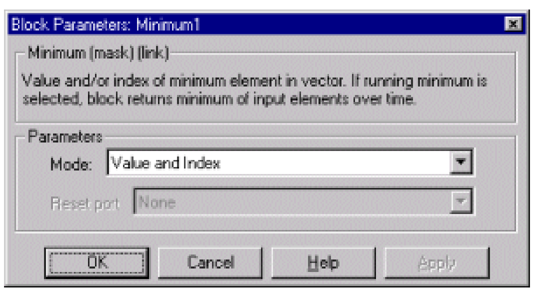
- **Delay (samples)** = `2`
- **Sample time** = `1`

- **Samples per frame** = 1

The block's operation is shown in the figure below.



Dialog Box



Mode

The block's mode of operation: Output the minimum value of each input (**Value**), the index of the minimum value (**Index**), both the value and the index (**Value and Index**), or track the minimum values in the input sequence over time (**Running**).

Reset port

Determines the reset event that causes the block to reset the running minimum. 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 more information, see "Resetting the Running Minimum" on page 7-410.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- 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 to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

Maximum	DSP Blockset
Mean	DSP Blockset
MinMax	Simulink
Histogram	DSP Blockset
`min`	MATLAB

Also see "Statistics" on page 7-18 for a list of all the blocks in the Statistics library.

Modified Covariance AR Estimator

Purpose Compute an estimate of AR model parameters using the 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.

$$H(z) = \frac{G}{A(z)} = \frac{G}{1 + a(2)z^{-1} + \ldots + a(p+1)z^{-p}}$$

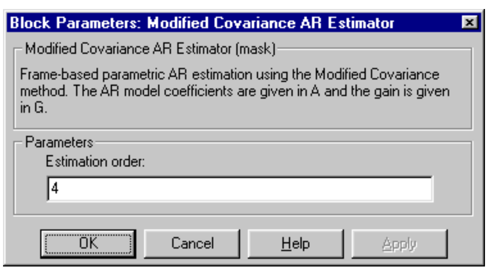
The order, p, of the all-pole model is specified by the **Order** parameter.

The top output, `A`, 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`).

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

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

To learn how to convert to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Burg AR Estimator	DSP Blockset
Covariance AR Estimator	DSP Blockset
Modified Covariance Method	DSP Blockset
Yule-Walker AR Estimator	DSP Blockset
`armcov`	Signal Processing Toolbox

Also see “Parametric Estimation” on page 7-5 for a list of all the blocks in the Parametric Estimation library.

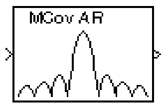
Purpose

Compute a parametric spectral estimate using the 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, and 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 **Inherit FFT length from input dimensions** is selected, N_{fft} is specified by the frame size of the input, which must be a power of 2. When **Inherit FFT length from input dimensions** is *not* selected, 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.

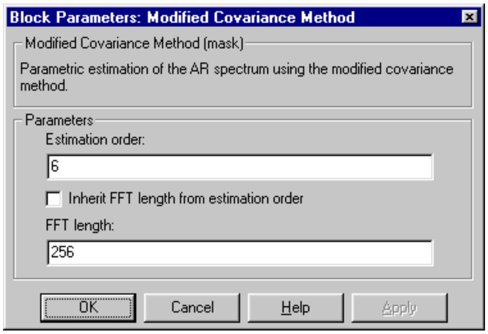
See the Burg Method block reference for a comparison of the Burg Method, Covariance Method, Modified Covariance Method, and Yule-Walker Method blocks.

Examples

The `dspsacomp` demo compares the modified covariance method with several other spectral estimation methods.

Modified Covariance Method

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. If N_{fft} exceeds the input frame size, the frame is zero-padded as needed. This parameter is enabled when **Inherit FFT length from input dimensions** is not selected.

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

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

To learn how to convert to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Burg Method	DSP Blockset
Covariance Method	DSP Blockset
Modified Covariance AR Estimator	DSP Blockset
Short-Time FFT	DSP Blockset
Yule-Walker Method	DSP Blockset
`pmcov`	Signal Processing Toolbox

See "Power Spectrum Estimation" on page 6-6 for related information. Also see a list of all blocks in the Power Spectrum Estimation library.

Multiphase Clock

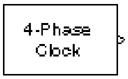
Purpose

Generate multiple binary clock signals.

Library

- DSP Sources
- Signal Management / Switches and Counters

Description



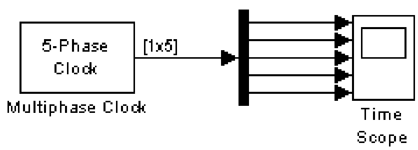
The Multiphase Clock block generates a sample-based 1-by-N vector of clock signals, where the integer N is specified by the **Number of phases** parameter. Each of the N phases has the same frequency, *f*, specified in hertz by the **Clock frequency** parameter.

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.

Example

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.

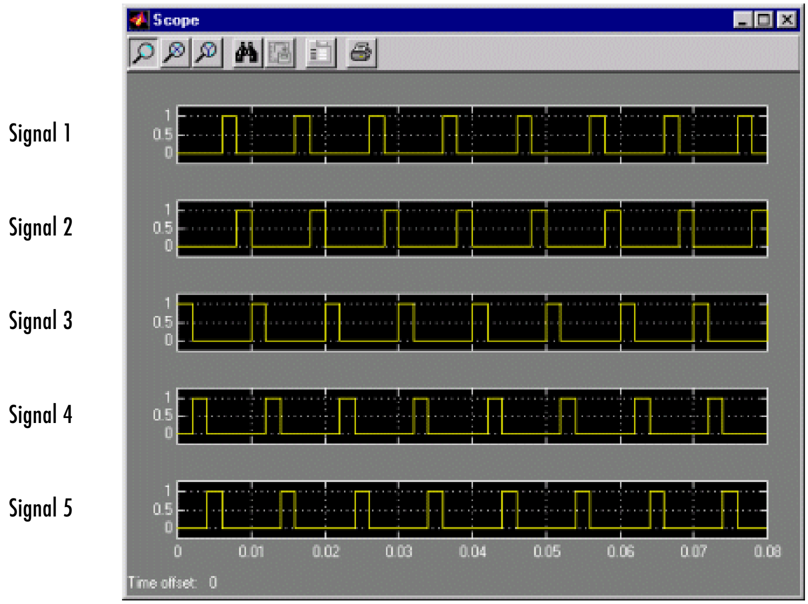


The corresponding settings are as follows:

- **Clock frequency** = 100
- **Number of phases** = 5
- **Starting phase** = 3
- **Number of phase intervals over which the clock is active** = 1

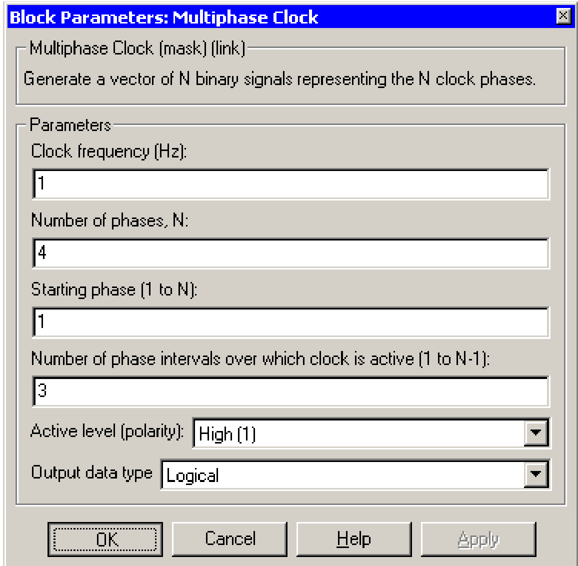
- **Active level (polarity) = High (1)**

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



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.

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 A-11.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed-point
- Boolean — The block may output Boolean values depending on the **Output data type** parameter setting, as described in "Effects of Enabling and Disabling Boolean Support" on page A-11. To learn how to disable Boolean output support, see "Steps to Disabling Boolean Support" on page A-12.

To learn how to convert to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

Clock	Simulink
Counter	DSP Blockset
Pulse Generator	Simulink
Event-Count Comparator	DSP Blockset

Also see the following topics:

- "Creating Signals Using Signal Generator Blocks" on page 3-36 — How to use this and other blocks to generate signals
- "DSP Sources" on page 7-3 — List of all blocks in the DSP Sources library
- "Switches and Counters" on page 7-15 — List of all blocks in the Switches and Counters library.

Multiport Selector

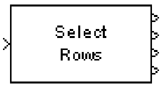
Purpose

Distribute arbitrary subsets of input rows or columns to multiple output ports.

Library

Signal Management / Indexing

Description



The Multi-port 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 kth cell contains a one-dimensional indexing expression specifying the subset of input rows or columns to be propagated to the kth 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.

The **Indices to output** parameter is tunable, so you can change the values of the indices at any time during the simulation; however, the number of cells in the array (i.e., the number of output ports) and the size of each submatrix in the output must remain the same while the simulation is running.

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.

Example

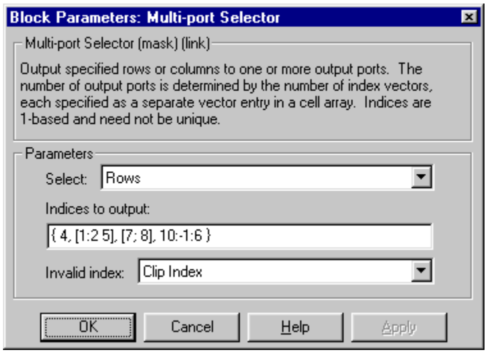
Consider the following **Indices to output** cell array:

```
{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. This parameter is tunable, but the size of the cell array (i.e., the number of output ports) and the size of each submatrix in the output must remain the same while the simulation is running.

Invalid index

Response to an invalid index value. Tunable.

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 to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Permute Matrix	DSP Blockset
Selector	Simulink
Submatrix	DSP Blockset
Variable Selector	DSP Blockset

Also see “Indexing” on page 7-14 for a list of all the blocks in the Indexing library.

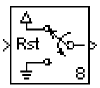
Purpose

Output ones or zeros for a specified number of sample times

Library

- DSP Sources
- Signal Management / Switches and Counters

Description



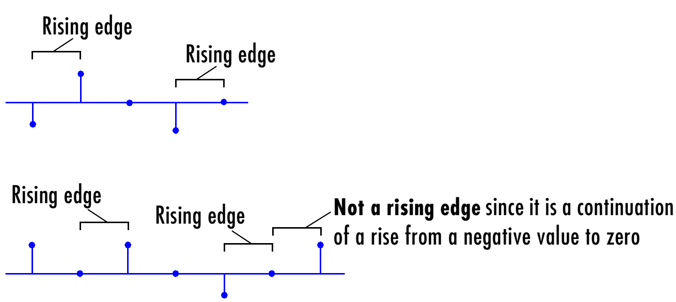
The N-Sample Enable block outputs the *inactive* value (`0` or `1`, whichever is *not* selected in the **Active level** parameter) during the first N sample times, where N is the **Trigger count** value. Beginning with output sample N+1, the block outputs the *active* value (`1` or `0`, whichever is selected 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.

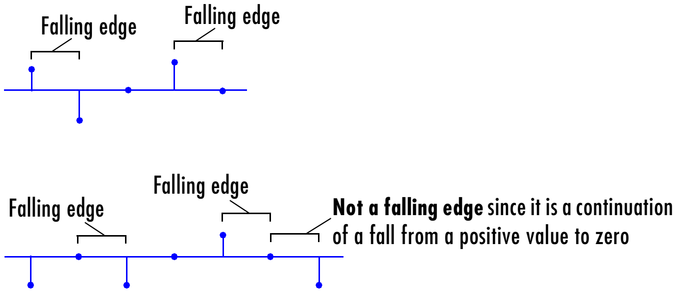
The triggering event is specified by the **Trigger type** pop-up menu, and can be one of the following:

- **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

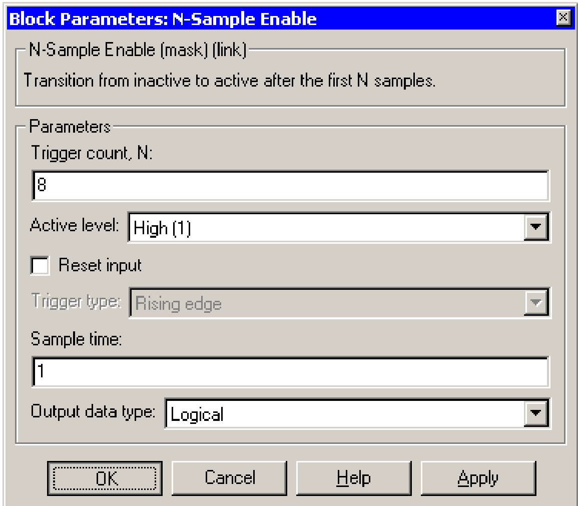
- Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero (see the following figure)



- **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 3-91 and the topic on the **Simulation Parameters** dialog box in the Simulink documentation.

Dialog Box



Trigger count

The number of samples for which the block outputs the active value. Tunable.

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. Tunable.

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. For information on the **Logical** and **Boolean** options of this parameter, see "Effects of Enabling and Disabling Boolean Support" on page A-11.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed-point
- Boolean — The block accepts Boolean inputs to the `Rst` port, which is enabled when you set the **Reset input** parameter. The block may output Boolean values depending on the **Output data type** parameter setting, as described in "Effects of Enabling and Disabling Boolean Support" on page A-11. To learn how to disable Boolean output support, see "Steps to Disabling Boolean Support" on page A-12.

To learn how to convert to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

Counter	DSP Blockset
N-Sample Switch	DSP Blockset

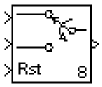
Also see the following topics:

- "Creating Signals Using Signal Generator Blocks" on page 3-36 — How to use this and other blocks to generate signals
- "DSP Sources" on page 7-3 — List of all blocks in the DSP Sources library
- "Switches and Counters" on page 7-15 — List of all blocks in the Switches and Counters library.

Purpose Switch between two inputs after a 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 N is specified by the **Switch count** value. 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.

The sample period of the output is specified by the **Sample time** parameter (i.e., 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.

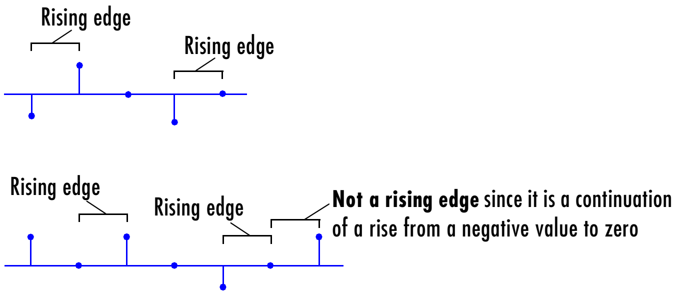
The inputs must either both be frame-based or both be sample-based.

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.

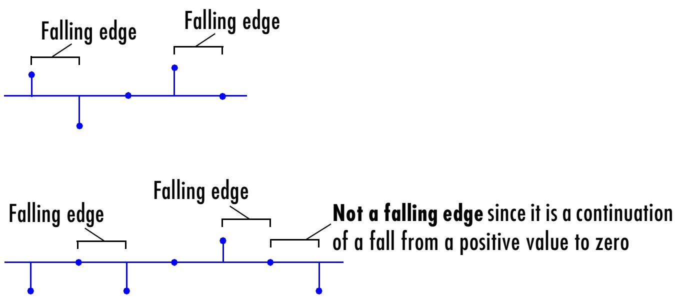
The triggering event is specified by the **Trigger type** pop-up menu, and can be one of the following:

- **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)

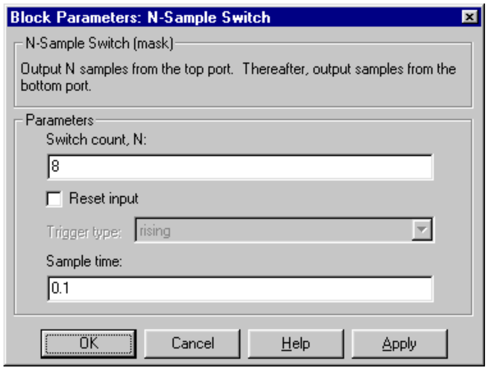


- **Either edge** — Triggers a reset operation when the `Rst` input is a **Rising edge** or **Falling edge** (as described above).
- **Non-zero sample** — Triggers a reset operation at each sample time that the `Rst` input is not zero.

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 3-91 and the topic on the **Simulation Parameters** dialog box in the Simulink 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 **Reset input** is selected. 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
- Boolean — The block accepts Boolean inputs to the `Rst` port, which is enabled when you set the **Reset input** parameter.

To learn how to convert to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

Counter	DSP Blockset
N-Sample Enable	DSP Blockset

Also see "Switches and Counters" on page 7-15 for a list of all the blocks in the Switches and Counters library.

Normalization

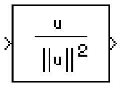
Purpose

Normalize an input by its 2-norm or squared 2-norm.

Library

Math Functions / Math Operations

Description



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 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

When the **Norm** parameter specifies **2-norm**, the block normalizes the jth input column as follows.

$$y_{ij} = \frac{u_{ij}}{\|u\|_j + b}$$

where b is specified by the **Normalization bias** parameter, and $\|u\|_j$ is the 2-norm (or *Euclidean* norm) of the jth input column.

$$\|u\|_j = \sqrt{|u_1|^2 + |u_2|^2 + \cdots + |u_M|^2}$$

Equivalently,

```
y = u ./ (norm(u) + b)      % Equivalent MATLAB code
```

The normalization bias, b, is typically chosen to be a small positive constant (e.g., 1e-10) that prevents potential division by zero.

Squared 2-Norm

When the **Norm** parameter specifies **Squared 2-norm**, the block normalizes the jth input column as follows.

$$y_{ij} = \frac{u_{ij}}{\|u\|_j^2 + b}$$

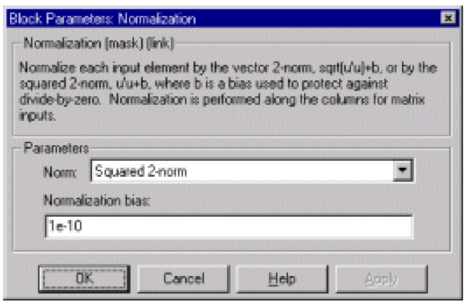
where

$$\|u\|_j^2 = |u_1|^2 + |u_2|^2 + \cdots + |u_M|^2$$

Equivalently,

```
y = u ./ (norm(u).^2 + b)   % Equivalent MATLAB code
```

Dialog Box



Norm

The type of normalization to apply, **2-norm** or **Squared 2-norm**. Tunable in simulation.

Normalization bias

The real value b to be added in the denominator to avoid division by zero. Tunable in simulation.

Supported Data Types

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

To learn how to convert to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Matrix Scaling	DSP Blockset
Reciprocal Condition	DSP Blockset
`norm`	MATLAB

Also see "Math Operations" on page 7-9 for a list of all the blocks in the Math Operations library.

Overlap-Add FFT Filter

Purpose Implement the overlap-add method of frequency-domain filtering.

Library Filtering / Filter Designs

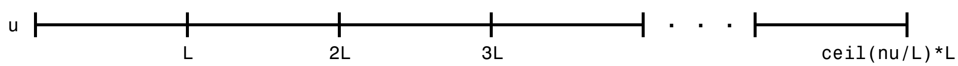
Description The Overlap-Add FFT Filter block uses an FFT to implement the *overlap-add method*, a technique that combines successive frequency-domain filtered sections of an input sequence.

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.

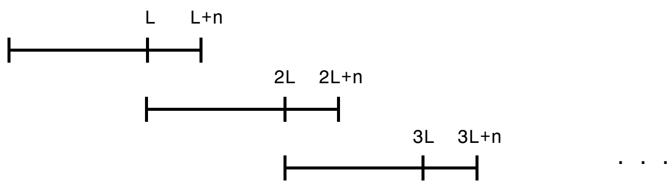
If either the filter coefficients or the inputs to the block are complex, the **Output** parameter should be set to **Complex**. Otherwise, the default **Output** setting, **Real**, instructs the block to take only the real part of the solution.

The block's overlap-add operation is equivalent to

```
y = ifft(fft(u(i:i+L-1),nfft) .* fft(b,nfft))
```

where `nfft` is specified by 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
```

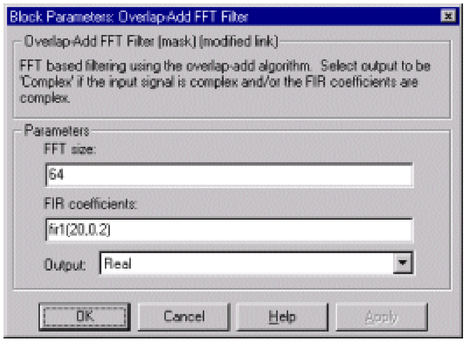
Latency

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`.

See “Excess Algorithmic Delay (Tasking Latency)” on page 3-91 and “The Simulation Parameters Dialog Box” in the Simulink documentation for more information about block rates and the Simulink tasking modes.

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**. If 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.

Supported Data Types

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

To learn how to convert to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Overlap-Save FFT Filter	DSP Blockset

Also see “Filtering” on page 7-6 for a list of all DSP Blockset filtering blocks.

Purpose

Implement the 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.

If 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 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 `nfft` is specified by 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.

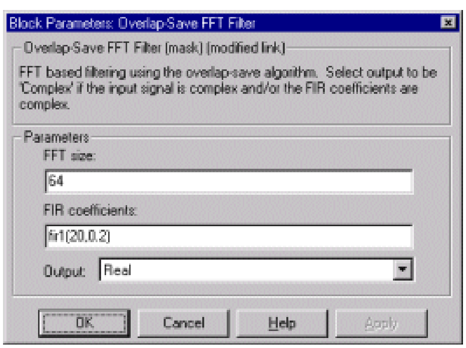
Latency

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`.

See “Excess Algorithmic Delay (Tasking Latency)” on page 3-91 and “The Simulation Parameters Dialog Box” in the Simulink documentation for more information about block rates and the Simulink tasking modes.

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**. If 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.

Supported Data Types

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

To learn how to convert to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Overlap-Add FFT Filter	DSP Blockset

Also see “Filtering” on page 7-6 for a list of all DSP Blockset filtering blocks.

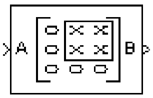
Overwrite Values

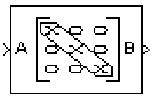
Purpose Overwrite a submatrix or subdiagonal of the input.

Library

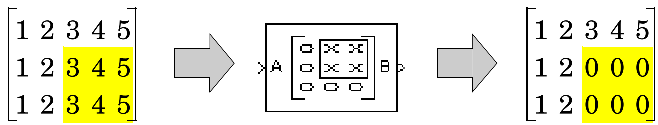
- Math Functions / Matrices and Linear Algebra / Matrix Operations
- Signal Management / Indexing

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 (useful for providing overwriting values that change at each time step).



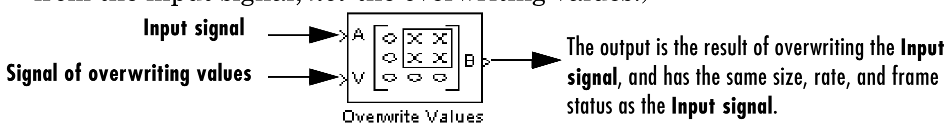
The block accepts both sample- and frame-based vectors and matrices. The output has the same size and frame status as the original input signal (not necessarily the same size and frame status as the signal containing the overwriting values).

Sections of This Reference Page

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 7-445.
- **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.)

  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:

  - If both signals are sample based, their sample rates must be the same.
  - If both signals are frame based, their frame rates must be the same.
  - If 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.

Table 7-15: Valid Overwriting Values

Portion of Input to Overwrite	Valid Overwriting Values	Example
A single element in the input $\begin{bmatrix} x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \end{bmatrix}$	Any constant value, v	$v = 9 \quad \begin{bmatrix} x & x & x & x & x \\ x & x & x & 9 & x \\ x & x & x & x & x \\ x & x & x & x & x \end{bmatrix}$
A length-k portion of the diagonal $\begin{bmatrix} x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \end{bmatrix}$	Any length-k column or row vector, v	$k = 3 \quad v = \begin{bmatrix} 2 & 4 & 6 \end{bmatrix}$ or $\begin{bmatrix} 2 \\ 4 \\ 6 \end{bmatrix}$ $\begin{bmatrix} 2 & x & x & x & x \\ x & 4 & x & x & x \\ x & x & 6 & x & x \\ x & x & x & x & x \end{bmatrix}$
A length-k portion of a row $\begin{bmatrix} x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \end{bmatrix}$	Any length-k row vector, v	$k = 3 \quad v = \begin{bmatrix} 2 & 4 & 6 \end{bmatrix}$ $\begin{bmatrix} x & x & x & x & x \\ x & 2 & 4 & 6 & x \\ x & x & x & x & x \\ x & x & x & x & x \end{bmatrix}$

Table 7-15: Valid Overwriting Values (Continued)

Portion of Input to Overwrite	Valid Overwriting Values	Example
A length-k portion of a column $\begin{bmatrix} x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \end{bmatrix}$	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 $\begin{bmatrix} x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \end{bmatrix}$	Any m-by-n matrix, v	$m = 2 \quad n = 3 \quad v = \begin{bmatrix} 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$ $\begin{bmatrix} x & x & x & x & x \\ x & x & 4 & 5 & 6 \\ x & x & 7 & 8 & 9 \\ x & x & x & x & x \end{bmatrix}$

Overwriting a Submatrix

To overwrite a submatrix, do the following:

1. Set the **Overwrite** parameter to **Submatrix**.
2. Specify the overwriting values as described in "Specifying the Overwriting Values" on page 7-445.
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. If 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, if 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 Table 7-16, Settings for Row, Column, Starting Row, and Starting Column Parameters, on page 7-448 and Table 7-17, Settings for Ending Row and Ending Column Parameters, on page 7-450.

Table 7-16: 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

Table 7-16: Settings for Row, Column, Starting Row, and Starting Column Parameters (Continued)

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 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
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

Table 7-17: 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
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

Overwriting a Subdiagonal

To overwrite a subdiagonal, do the following:

1. Set the **Overwrite** parameter to **Diagonal**.
2. Specify the overwriting values as described in "Specifying the Overwriting Values" on page 7-445.
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 If 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 Table 7-16, Settings for Row, Column, Starting Row, and Starting Column Parameters, on page 7-448 and Table 7-17, Settings for Ending Row and Ending Column Parameters, on page 7-450.

Table 7-18: Element and Starting Element Parameters

Settings for Element and Starting Element Parameters	**First Element in Subdiagonal** (Only element if **Diagonal span** = **One element**)
First	Diagonal element in first row of the input
Index	*k*th diagonal element, where *k* is the value of the **Element index** or **Starting 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 **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

Table 7-19: Ending Element Parameter

Settings for Ending Element Parameter	Last Element in Subdiagonal
Index	*k*th 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
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

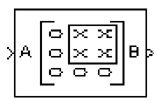
Example

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
- **Ending column = Last**

The figure below shows the block with the above settings overwriting a portion of a 3-by-5 input matrix.

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \end{bmatrix}$$

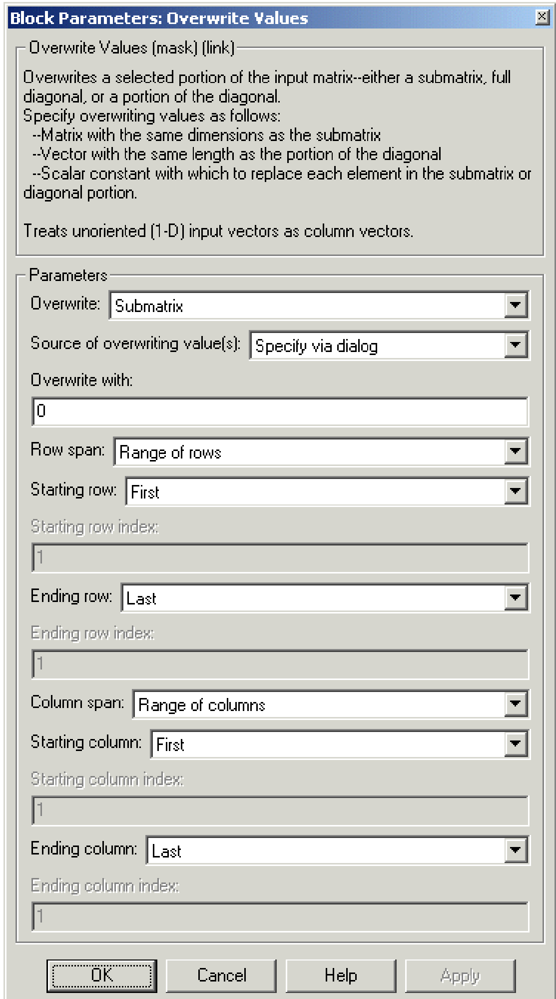


$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 0 & 0 & 0 \\ 1 & 2 & 0 & 0 & 0 \end{bmatrix}$$

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

Dialog Box



Note Only some of the following parameters are visible in the dialog box at any one time.

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 7-445.

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 7-445.

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 "Overwriting a Submatrix" on page 7-448.

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 Table 7-16, Settings for Row, Column, Starting Row, and Starting Column Parameters, on page 7-448. (**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 Table 7-16, Settings for Row, Column, Starting Row, and Starting Column Parameters, on page 7-448. (**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 Table 7-16, Settings for Row, Column, Starting Row, and Starting Column Parameters, on page 7-448. (**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 Table 7-17, Settings for Ending Row and Ending Column Parameters, on page 7-450. (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 Table 7-17, Settings for Ending Row and Ending Column Parameters, on page 7-450. (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 Table 7-17, Settings for Ending Row and Ending Column Parameters, on page 7-450. (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 "Overwriting a Submatrix" on page 7-448.

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 Table 7-16, Settings for Row, Column, Starting Row, and Starting Column Parameters, on page 7-448. (**Column** is enabled when **Column span** is set to **One column**, and **Starting column** when **Column span** is set to **Range of columns**.)

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 Table 7-16, Settings for Row, Column, Starting Row, and Starting Column Parameters, on page 7-448. (**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 Table 7-16, Settings for Row, Column, Starting Row, and Starting Column Parameters, on page 7-448. (**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 Table 7-17, Settings for Ending Row and Ending Column Parameters, on page 7-450. (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 Table 7-17, Settings for Ending Row and Ending Column Parameters, on page 7-450. (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 Table 7-17, Settings for Ending Row and Ending Column Parameters, on page 7-450. (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 7-450.

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 Table 7-18, Element and Starting

Element Parameters, on page 7-451. (**Element** is enabled when **Element span** is set to **One element**, and **Starting element** when **Element span** is set to **Range of element**s.)

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 Table 7-18, Element and Starting Element Parameters, on page 7-451. (**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 Table 7-18, Element and Starting Element Parameters, on page 7-451. (**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 Table 7-19, Ending Element Parameter, on page 7-452. (Enabled when **Element span** is set to **Range of element**s, 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 Table 7-19, Ending Element Parameter, on page 7-452. (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 Table 7-19, Ending Element Parameter, on page 7-452. (Enabled when **Ending element** is set to **Offset from middle** or **Offset from last**.)

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 to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Reshape	Simulink
Selector	Simulink
Submatrix	DSP Blockset
Variable Selector	DSP Blockset
`reshape`	MATLAB

Also see “Matrix Operations” on page 7-11 and “Indexing” on page 7-14 for lists of all the blocks in these libraries.

Pad

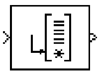
Purpose

Alter the input size by padding or truncating rows and/or columns.

Library

Signal Operations

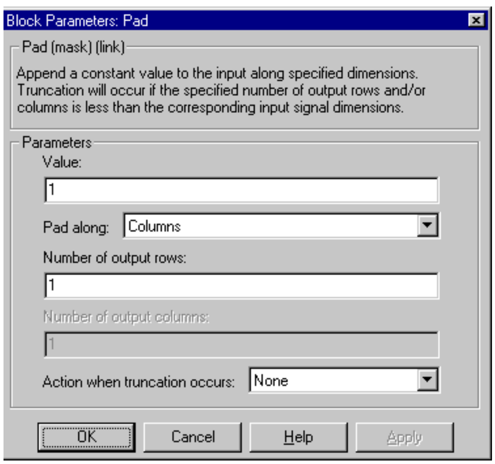
Description



The Pad block changes the size of the input matrix from M_i-by-N_i to M_o-by-N_o by padding or truncating along the rows, the columns, or both dimensions. The dimensions of the output, M_o and N_o, are specified by the **Number of output rows** and **Number of output columns** parameters, respectively. The value with which to pad the input is set by the **Value** parameter.

The behavior of the Pad block and Zero Pad block are identical, with the exception that the Pad block can pad the input matrix with values other than zero. See the Zero Pad block reference for more information on the behavior of the Pad block.

Dialog Box



Value

The scalar value with which to pad the input matrix. Tunable.

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 desired number of rows in the output, M_o. This parameter is enabled when **Columns** or **Columns and rows** is selected in the **Pad along** menu.

Number of output columns

The desired number of columns in the output, N_o. This parameter is enabled when **Rows** or **Columns and rows** is selected in the **Pad along** menu.

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 to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Matrix Concatenation	Simulink
Repeat	DSP Blockset
Submatrix	DSP Blockset
Upsample	DSP Blockset
Variable Selector	DSP Blockset
Zero Pad	DSP Blockset

Also see “Signal Operations” on page 7-16 for a list of all the blocks in the Signal Operations library.

Permute Matrix

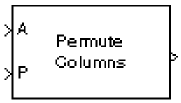
Purpose

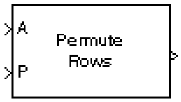
Reorder the rows or columns of a matrix.

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:

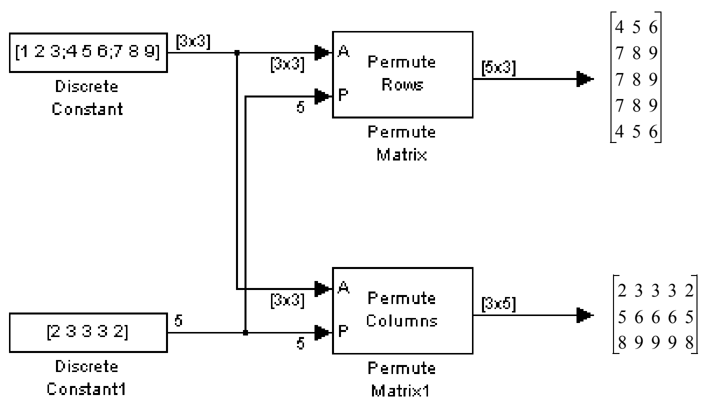
- **Clip index** — Clip the index to the nearest valid value (1 or M for row permutation, and 1 or N for column permutation), and *do not* issue an alert. Example: For a 3-by-7 input matrix, a column index of 9 is clipped to 7, and a row index of -2 is clipped to 1.
- **Clip and warn** — Display a warning 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 checking **Error when length of P is not equal to Permute dimension size**.

If input A is frame-based, the output is frame-based; otherwise, the output is sample-based.

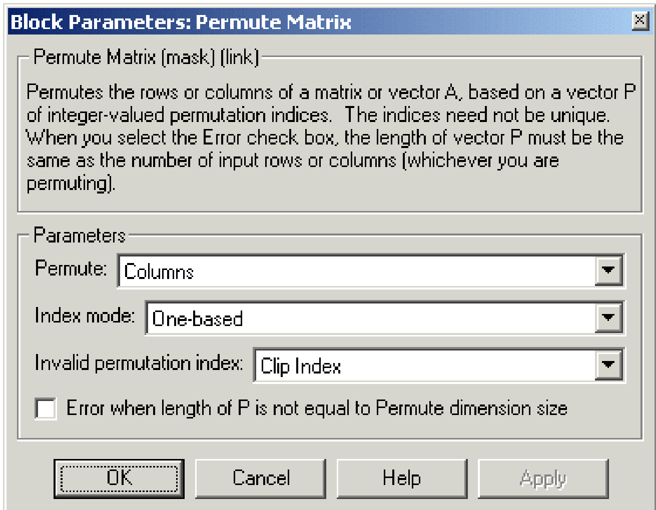
Example

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, except in the Simulink external mode.

Error when length of P is not equal to Permute dimension size

Option to display an error dialog box and terminate the simulation if the length of the permutation vector `P` is not equal to the number of rows or columns of the input matrix `A`.

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 to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Submatrix	DSP Blockset
Transpose	DSP Blockset
Variable Selector	DSP Blockset
`permute`	MATLAB

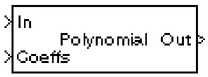
See “Reordering Channels in a Frame-Based Multichannel Signal” on page 3-61 for related information. Also see “Matrix Operations” on page 7-11 for a list of all the blocks in the Matrix Operations library.

Polynomial Evaluation

Purpose Evaluate a 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.

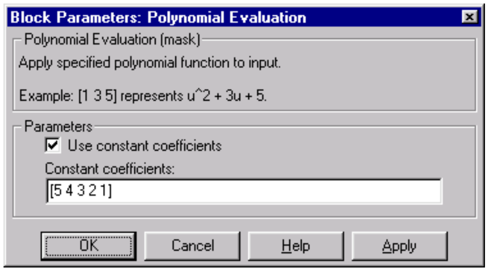
When the **Use constant coefficients** check box is selected, the polynomial expression is specified by the **Constant coefficients** parameter. When **Use constant coefficients** is not selected, a variable polynomial expression is specified by the input to the `Coeffs` port. In both cases, the polynomial 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



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 the **Use constant coefficients** check box is selected.

Supported Data Types

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

To learn how to convert to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Least Squares Polynomial Fit	DSP Blockset
Math Function	Simulink
Sum	Simulink
`polyval`	MATLAB

Also see “Polynomial Functions” on page 7-12 for a list of all the blocks in the Polynomial Functions library.

Polynomial Stability Test

Purpose Determine whether all roots of the input polynomial are inside the unit circle using the Schur-Cohn algorithm.

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.


```
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; i.e., 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 may be unstable; i.e., the magnitude of at least one solution to $f(x) = 0$ is greater than or equal to 1.

The output is always sample-based.

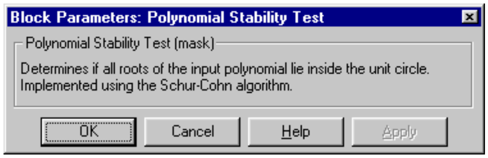
Applications

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, $A(z)$. If 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



Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Boolean — Block outputs are always Boolean.

To learn how to convert to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

Least Squares Polynomial Fit	DSP Blockset
Polynomial Evaluation	DSP Blockset
`polyfit`	MATLAB

Also see "Polynomial Functions" on page 7-12 for a list of all the blocks in the Polynomial Functions library.

Pseudoinverse

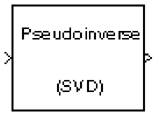
Purpose

Compute the Moore-Penrose pseudoinverse of a 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

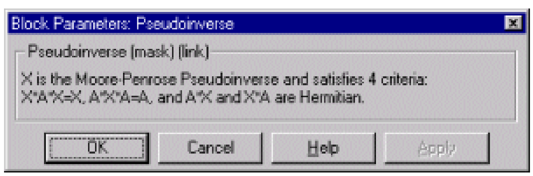
$$A^+ = VS^+U^*$$

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



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 to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Cholesky Inverse	DSP Blockset
LDL Inverse	DSP Blockset
LU Inverse	DSP Blockset
Singular Value Decomposition	DSP Blockset
`inv`	MATLAB

See "Inverting Matrices" on page 6-10 for related information. Also see "Matrix Inverses" on page 7-11 for a list of all the blocks in the Matrix Inverses library.

QR Factorization

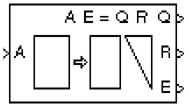
Purpose

Factor a rectangular matrix into unitary and upper triangular components.

Library

Math Functions / Matrices and Linear Algebra / Matrix Factorizations

Description



The QR Factorization block uses 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.

$$|r_{i+1,j+1}| > |r_{i,j}| \qquad i = j$$

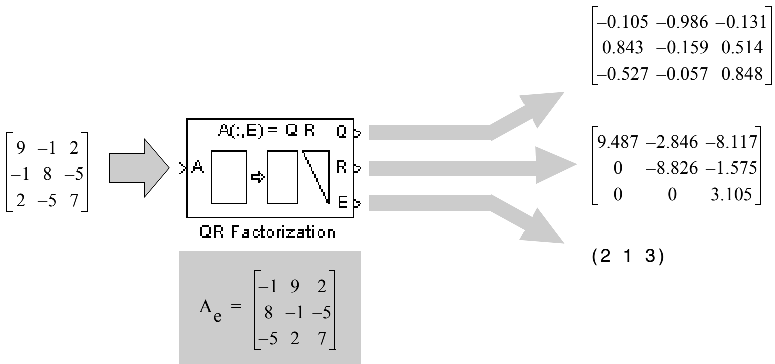
QR factorization is an important tool for solving linear systems of equations because of good error propagation properties and the invertability of unitary matrices.

$$Q^{-1} = Q^{*}$$

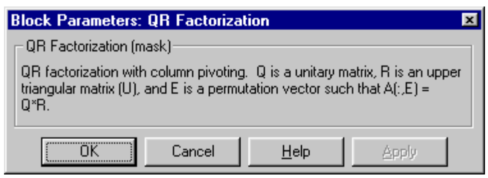
Unlike LU and Cholesky factorizations, the matrix A does not need to be square for QR factorization. Note, however, that QR factorization requires twice as many operations as Gaussian elimination.

Example

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 to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Cholesky Factorization	DSP Blockset
LU Factorization	DSP Blockset
QR Solver	DSP Blockset
Singular Value Decomposition	DSP Blockset
`qr`	MATLAB

See “Factoring Matrices” on page 6-8 for related information. Also see “Matrix Factorizations” on page 7-10 for a list of all the blocks in the Matrix Factorizations library.

QR Solver

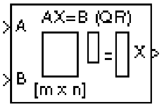
Purpose

Find a minimum-norm-residual solution to the equation AX=B.

Library

Math Functions / Matrices and Linear Algebra / Linear System Solvers

Description



The QR Solver block solves the linear system AX=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-hand-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

$$A_eX = B_e ,$$

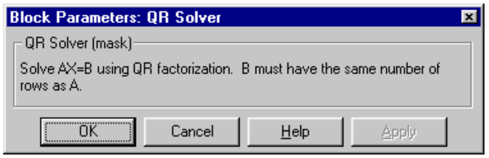
and

$$QRX = B_e$$

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



Supported Data Types

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

To learn how to convert to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Levinson-Durbin	DSP Blockset
LDL Solver	DSP Blockset
LU Solver	DSP Blockset
QR Factorization	DSP Blockset
SVD Solver	DSP Blockset

See “Solving Linear Systems” on page 6-7 for related information. Also see “Linear System Solvers” on page 7-9 for a list of all the blocks in the Linear System Solvers library.

Queue

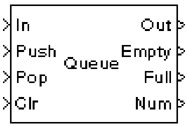
Purpose

Store inputs in a FIFO register.

Library

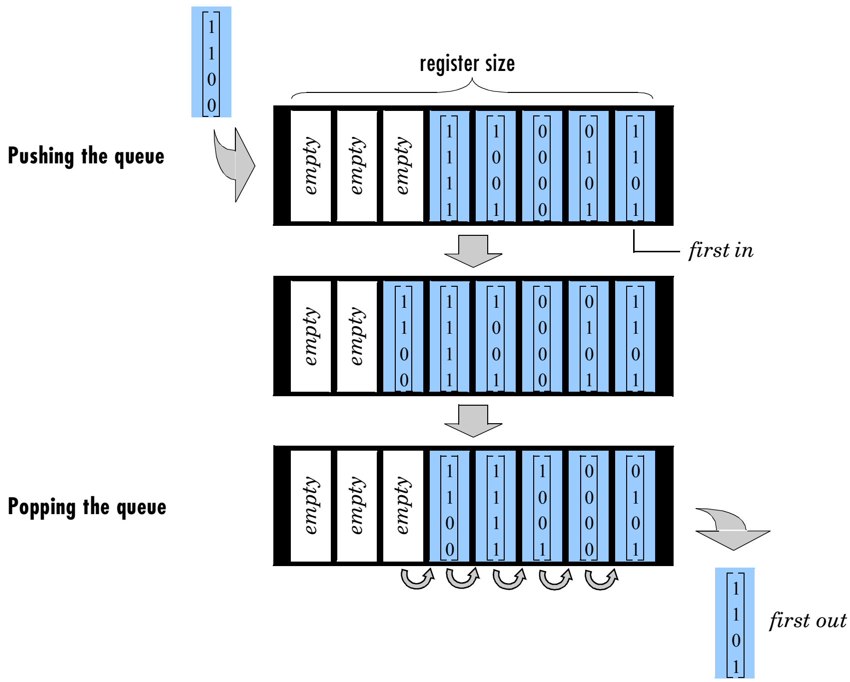
Signal Management / Buffers

Description



The Queue block stores a sequence of input samples in a FIFO (first in, first out) 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.



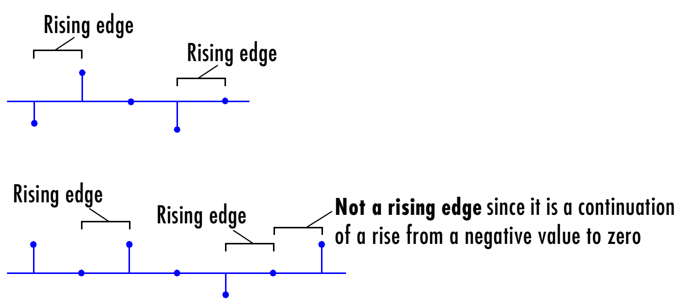
A trigger event at the optional `Clr` port (enabled by the **Clear input** check box) empties the queue contents. If **Clear output port on reset** is selected, then a trigger event at the `Clr` port empties the queue *and* sets the value at the `Out` port to zero. This setting also applies when a disabled subsystem containing the Queue block is reenabled; the `Out` port value is only reset to zero in this case if **Clear output port on reset** is selected.

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`

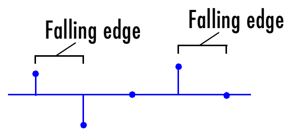
The rate of the trigger signal must be a positive integer multiple of the rate of the data signal input. The triggering event for the `Push`, `Pop`, `and Clr` ports is specified by the **Trigger type** pop-up menu, and can be one of the following.

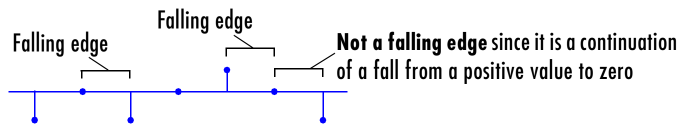
- **Rising edge** — Triggers execution of the block when the trigger input does one of the following:
  - Rises from a negative value to a positive value or zero
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)



- **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 3-91 and the topic on the **Simulation Parameters** dialog box in the Simulink documentation.

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.

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

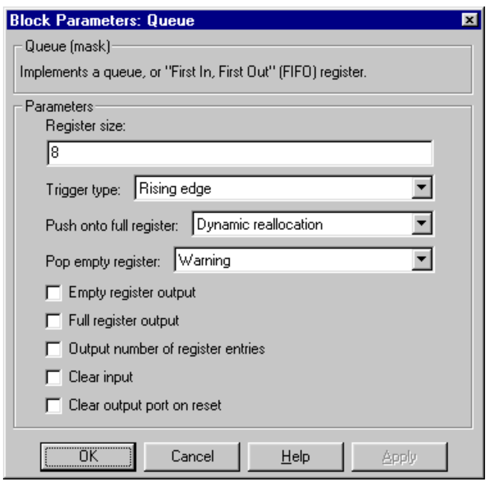
In	Push	Pop	Clr	Queue	Out	Empty	Full	Num
10	1	0	0	*top* [ ][ ][ ][10] *bottom*	5	0	0	1
11	0	0	0	*top* [ ][ ][11][10] *bottom*	5	0	0	2
12	1	0	1	*top* [ ][ ][ ][12] *bottom*	0	0	0	1

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 Queue operation.

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 a positive integer multiple of the rate of the data signal input. Tunable only in simulation (not tunable in Real-Time Workshop external mode or in the Simulink Performance Tools Accelerator).

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 **Dynamic reallocation** is selected 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.

Supported Data Types

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

- Custom data types
- Boolean — The block accepts Boolean inputs to the `Push`, `Pop`, and `Clr` ports. The block may output Boolean values at the `Out` and `Full` ports 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 A-11. To learn how to disable Boolean output support, see “Steps to Disabling Boolean Support” on page A-12.
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

To learn how to convert to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

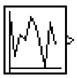
Buffer	DSP Blockset
Delay Line	DSP Blockset
Stack	DSP Blockset

Also see “Buffers” on page 7-14 for a list of all the blocks in the Buffers library.

Purpose Generate randomly distributed values.

Library DSP Sources

Description



The Random Source block generates a frame of M values drawn from a uniform or Gaussian pseudorandom distribution, where M is specified by the **Samples per frame** parameter.

This reference page contains a detailed discussion of the following Random Source block topics:

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 only one of the **Minimum** and **Maximum** parameters is specified as a vector, the other is scalar expanded to the same length.

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** — The block produces Gaussian random values by using the Ziggurat method, which is the same method used by the MATLAB `randn` function.
- **Sum of uniform values** — The block produces Gaussian random values by adding and scaling uniformly distributed random signals. 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 only one of these parameters is specified as a vector, the other is scalar expanded to the same length.

Output Complexity

The block's output can be either real or complex, as selected by the **Real** and **Complex** options in the **Output 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. If 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** — The block 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** — The block 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 the next section, "Specifying the Initial Seed".

- **Not repeatable** — The block 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

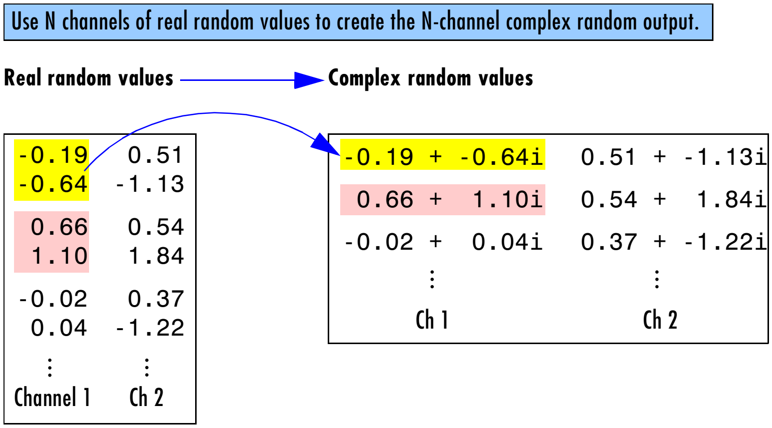
When you set the **Repeatability** parameter to **Specify seed**, you must set the **Initial seed** parameter. The **Initial seed** parameter specifies the initial seed for the pseudorandom number generator. The generator produces an identical sequence of pseudorandom numbers each time it is executed with a particular initial seed.

Specifying Initial Seeds for Real Outputs. To specify the N initial seeds for an N-channel real-valued output (**Output complexity** parameter set to **Real**), provide one of the following in the **Initial seed** parameter:

- Length-N vector of initial seeds — The block uses each vector element as an initial seed for the corresponding channel in the N-channel output.
- Single scalar — The block uses the scalar to generate N random values, which it uses as the seeds for the N-channel output.

Specifying Initial Seeds for Complex Outputs. To specify the initial seeds for an N-channel complex-valued output (**Output complexity** parameter set to **Complex**), provide one of the following in the **Initial seed** parameter:

- Length-N vector of initial seeds — The block 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 — The block 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.



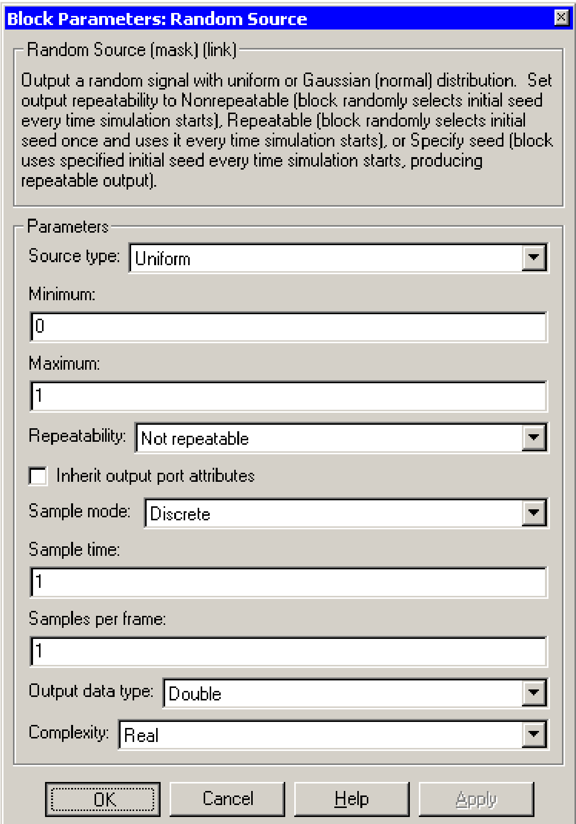
Sample Period

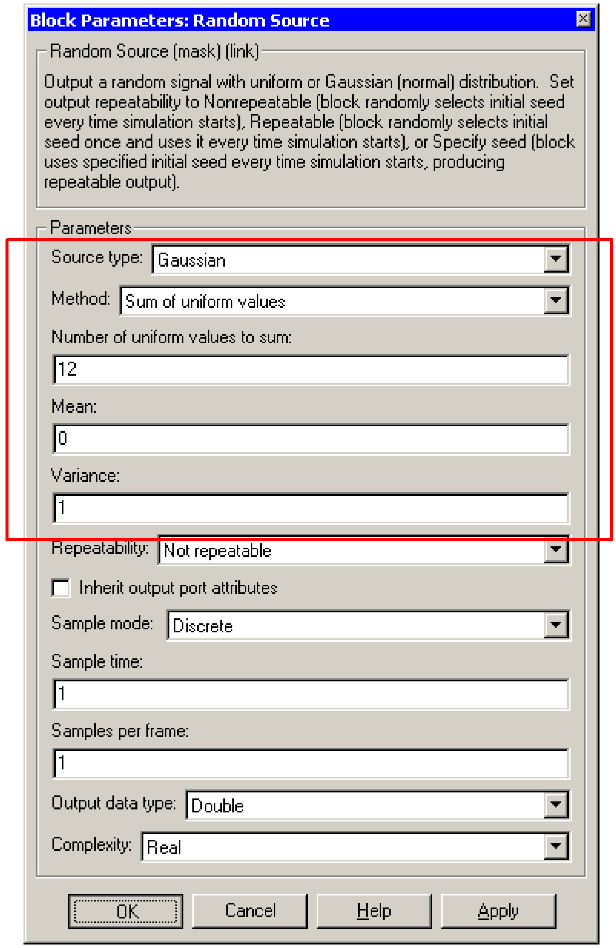
The **Sample time** parameter value, T_s, specifies the random sequence sample period when the **Sample mode** parameter is set to **Discrete**. In this mode, the block generates the number of samples specified by the **Samples per frame** parameter value, M, and outputs this frame with a period of $M*T_s$. 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.





Source type

The distribution from which to draw the random values, **Uniform** or **Gaussian**. For more information, see “Distribution Type” on page 7-483.

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 7-483.

Minimum

The minimum value in the uniform distribution. This parameter is enabled when **Uniform** is selected from the **Source type** parameter. Tunable.

Maximum

The maximum value in the uniform distribution. This parameter is enabled when **Uniform** is selected 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 7-483.

Mean

The mean of the Gaussian (normal) distribution. This parameter is enabled when **Gaussian** is selected from the **Source type** parameter. Tunable.

Variance

The variance of the Gaussian (normal) distribution. This parameter is enabled when **Gaussian** is selected 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 7-485.

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 7-486.

Inherit output port attributes

When selected, allows the block to inherit the sample mode, sample period, and complexity of a downstream block. (The **Sample mode**, **Sample time**, **Samples per frame**, and **Output complexity** parameters are disabled.) The output is a length-M sample-based 1-D vector, where length M is inherited from the downstream block. If 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 if M is not an integer multiple of N.

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 to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

Discrete Impulse	DSP Blockset
DSP Constant	DSP Blockset
Maximum	DSP Blockset
Minimum	DSP Blockset
Signal From Workspace	DSP Blockset
Standard Deviation	DSP Blockset
Variance	DSP Blockset
Random Number	Simulink
Signal Generator	Simulink
`rand`	MATLAB
`randn`	MATLAB

Also see the following topics:

- "Creating Signals Using Signal Generator Blocks" on page 3-36 — How to use this and other blocks to generate signals
- "DSP Sources" on page 7-3 — List of all blocks in the DSP Sources library

Real Cepstrum

Purpose Compute the real cepstrum of an 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 M_o is specified by 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
```

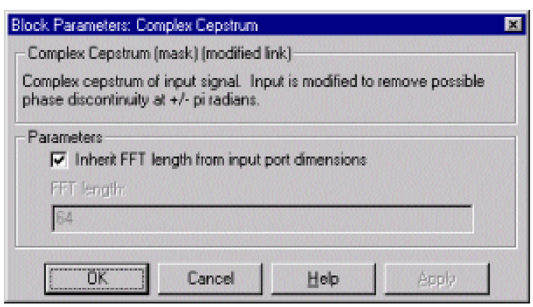
or, more compactly,

```
y = rceps(u,Mo)
```

When the **Inherit FFT length from input port dimensions** check box is selected, 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 (i.e., 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 **Inherit FFT length from input port dimensions** is not selected.

Supported Data Types

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

To learn how to convert to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

Complex Cepstrum	DSP Blockset
DCT	DSP Blockset
FFT	DSP Blockset
`rceps`	Signal Processing Toolbox

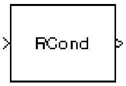
Also see "Transforms" on page 7-19 for a list of all the blocks in the Transforms library.

Reciprocal Condition

Purpose Compute the reciprocal condition of a 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

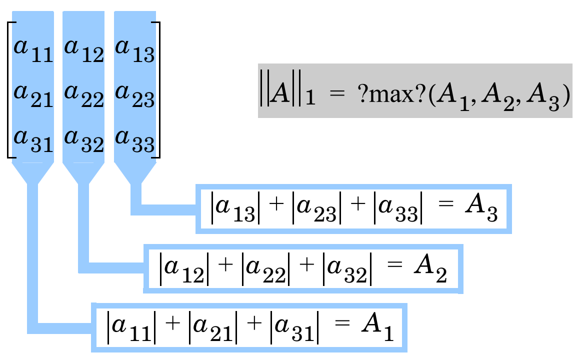
$$y = \frac{1}{\kappa} = \frac{1}{\|A^{-1}\|_1 \|A\|_1}$$

where κ is the condition number ($\kappa \geq 1$), and y is the scalar sample-based output ($0 \leq y < 1$).

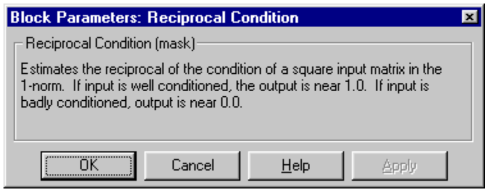
The matrix 1-norm, $\|A\|_1$, is the maximum column-sum in the M-by-M matrix A.

$$\|A\|_1 = ?\max?_{1 \leq j \leq M} \sum_{i=1}^{M} |a_{ij}|$$

For a 3-by-3 matrix:



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 to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Matrix 1-Norm	DSP Blockset
Normalization	DSP Blockset
`rcond`	MATLAB

Also see “Matrix Operations” on page 7-11 for a list of all the blocks in the Matrix Operations library.

Repeat

Purpose

Resample an input at a 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. The integer L is specified by the **Repetition count** parameter.

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.

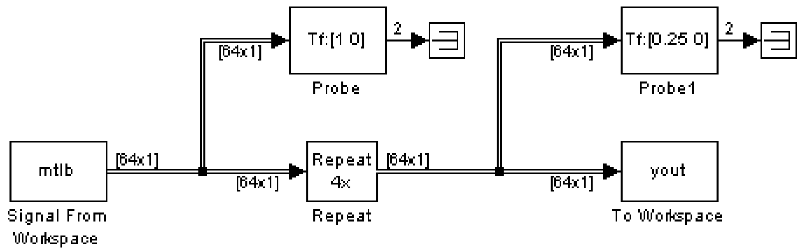
Frame-Based Operation

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.

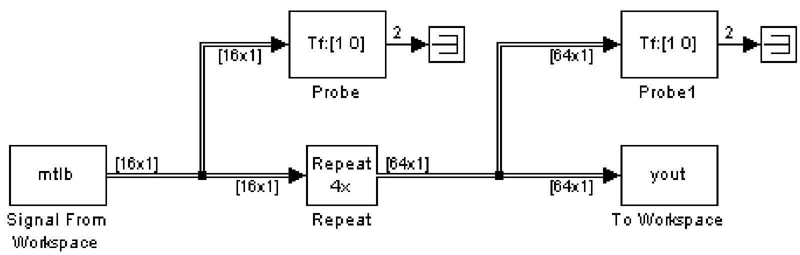
  The model below shows a single-channel input with a frame period of 1 second being upsampled through 4-times repetition to a frame period of 0.25 second. The input and output frame sizes are identical.



- **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.



Latency

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.

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.

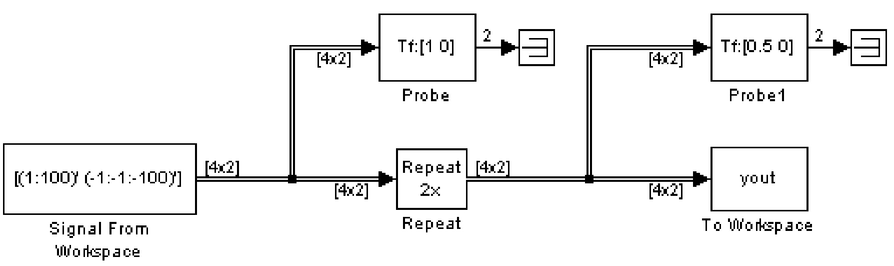
Nonzero Latency. The Repeat block has tasking latency only for multirate operation in the Simulink multitasking mode:

- 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.

See "Excess Algorithmic Delay (Tasking Latency)" on page 3-91 and "The Simulation Parameters Dialog Box" in the Simulink documentation for more information about block rates and the Simulink tasking modes.

Example

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, select **Fixed-step** and **discrete** from the **Type** controls in the **Solver** panel of the **Simulation Parameters** dialog box, and select **MultiTasking** from the **Mode** parameter. Also set the **Stop time** to `30`.

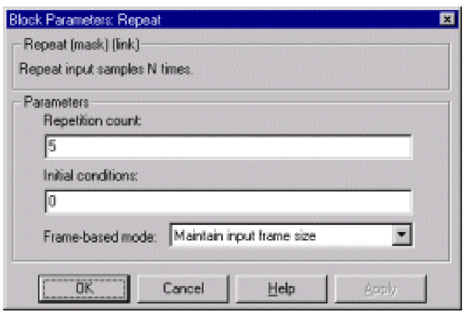
Run the model and look at the output, `yout`. The first few samples of each channel are shown below.

```
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 (i.e., 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. Tunable.

Initial conditions

The value with which the block is initialized for cases of nonzero latency; a scalar or matrix. Tunable.

Frame-based mode

For frame-based operation, the method by which to implement the repetition (upsampling): **Maintain input frame size** (i.e., increase the frame rate), or **Maintain input frame rate** (i.e., increase the frame size). The **Frame-based mode** parameter must be set to **Maintain input frame size** for sample-base inputs. Tunable.

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 to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

FIR Interpolation	DSP Blockset
Upsample	DSP Blockset
Zero Pad	DSP Blockset

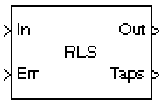
Also see “Signal Operations” on page 7-16 for a list of all the blocks in the Signal Operations library.

RLS Adaptive Filter

Purpose Compute filter estimates for an input using the RLS adaptive filter algorithm.

Library Filtering / Adaptive Filters

Description The RLS Adaptive Filter block recursively computes the least squares estimate (RLS) of the FIR filter coefficients.



The corresponding RLS filter is expressed in matrix form as

$$
\begin{aligned}
k(n) &= \frac{\lambda^{-1}P(n-1)u(n)}{1+\lambda^{-1}u^{H}(n)P(n-1)u(n)} \\
y(n) &= \hat{w}^{H}(n-1)u(n) \\
e(n) &= d(n)-y(n) \\
\hat{w}(n) &= \hat{w}(n-1)+k(n)e^{*}(n) \\
P(n) &= \lambda^{-1}P(n-1)-\lambda^{-1}k(n)u^{H}(n)P(n-1)
\end{aligned}
$$

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
$y(n)$	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 the **Adapt input** check box is selected 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 "forgets" past sample information. Setting λ=1 specifies an infinite memory; typically, $0.95 \leq \lambda \leq 1$.

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

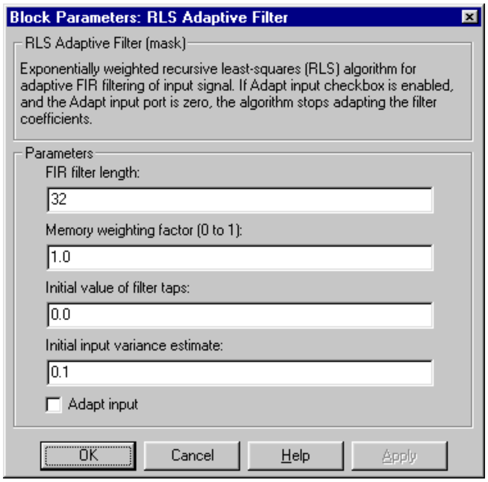
$$I\frac{1}{\hat{\sigma}^2}$$

where $\hat{\sigma}^2$ is specified by the **Initial input variance estimate** parameter.

Example

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 to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Kalman Adaptive Filter	DSP Blockset
LMS Adaptive Filter	DSP Blockset

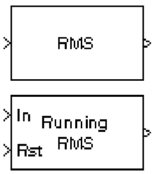
See "Adaptive Filters" on page 4-34 for related information. Also see a list of all blocks in the Adaptive Filters library.

RMS

Purpose Compute the 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 the **Running RMS** check box is *not* selected, 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))     % 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 RMS value for each column in `u`. The RMS value of the jth 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 the **Running RMS** check box is selected, 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 jth column over all inputs since the last reset, up to and including element u_{ij} of the current input.

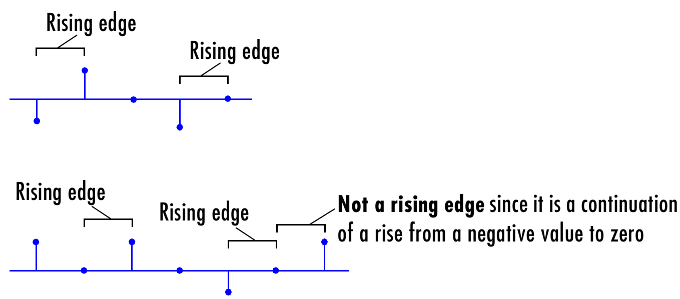
As in basic operation, length-M 1-D vector inputs and *sample-based* length-M row vector inputs are both treated as M-by-1 column vectors.

Resetting the Running 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.

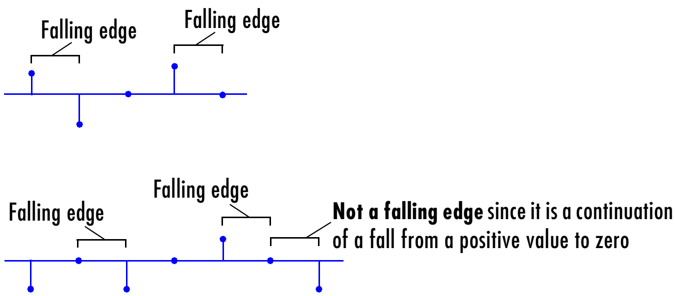
The reset event is specified by the **Reset port** parameter, and can be one of the following:

- **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)

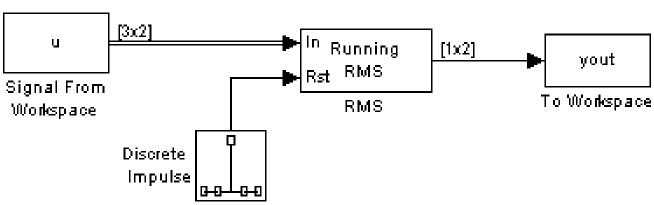


- **Either edge** — Triggers a reset operation when the `Rst` input is a **Rising edge** or **Falling edge** (as described above).
- **Non-zero sample** — Triggers a reset operation at each sample time that the `Rst` input is not zero.

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 3-91 and the topic on the **Simulation Parameters** dialog box in the Simulink documentation.

Example

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** = ☑
- **Reset port** = **Non-zero sample**

The Signal From Workspace block has the following settings:

- **Signal** = u
- **Sample time** = 1/3
- **Samples per frame** = 3

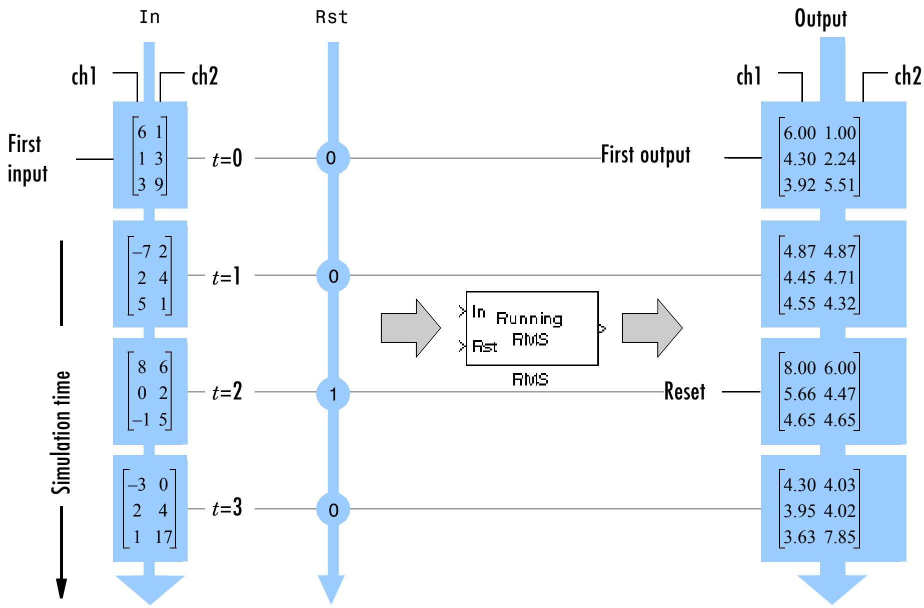
where

```
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]'
```

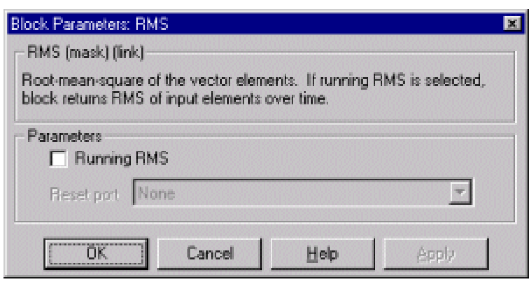
The Discrete Impulse block has the following settings:

- **Delay (samples)** = 2
- **Sample time** = 1
- **Samples per frame** = 1

The block's operation is shown in the figure below.



Dialog Box



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 7-507.

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 to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Mean	DSP Blockset
Variance	DSP Blockset

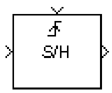
Also see “Statistics” on page 7-18 for a list of all the blocks in the Statistics library.

Sample and Hold

Purpose Sample and hold an 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. If the acquired input is frame-based, the output is frame-based; otherwise, the output is sample-based.

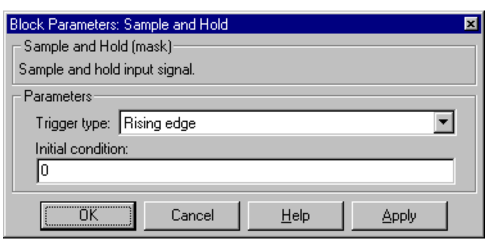


The trigger input must be a sample-based scalar with sample rate equal to the input frame rate at the signal port. The trigger event is specified by the **Trigger type** pop-up menu, and can be one of the following:

- **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.

The block's output prior to the first trigger event is specified by the **Initial condition** parameter. If 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. If 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.

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.

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 to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

Downsample	DSP Blockset
N-Sample Switch	DSP Blockset

Also see "Signal Operations" on page 7-16 for a list of all the blocks in the Signal Operations library.

Short-Time FFT

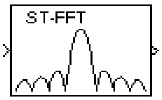
Purpose

Compute a nonparametric estimate of the spectrum using the short-time, fast Fourier transform (ST-FFT) method.

Library

Estimation / Power Spectrum Estimation

Description



The Short-Time FFT 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 **Inherit FFT length from input dimensions** is selected, N_{fft} is specified by the frame size of the input, which must be a power of 2. When **Inherit FFT length from input dimensions** is *not* selected, 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.

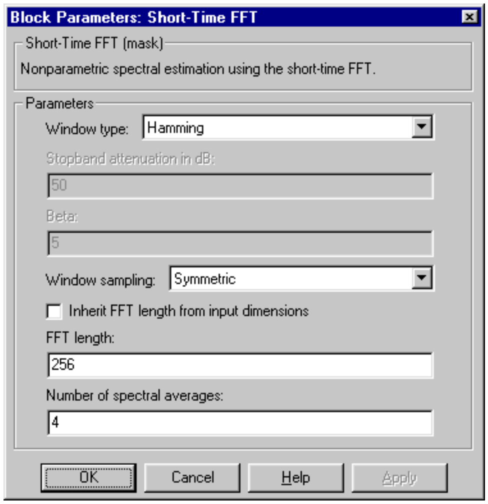
The **Number of spectral averages** specifies the number of spectra to average. Setting this parameter to `1` effectively disables averaging.

The **Window type**, **Stopband ripple**, **Beta**, and **Window sampling** parameters all apply to the specification of the window function; see the reference page for the Window Function block for more details on these four parameters.

Example

The `dspstfft` demo provides an illustration of using the Short-Time FFT and Matrix Viewer blocks to create a spectrogram. The `dspsacomp` demo compares the ST-FFT with several other spectral estimation methods.

Dialog Box



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. Disabled for other **Window type** selections. Tunable.

Beta

The β parameter for the Kaiser window. Disabled for other **Window type** selections. Increasing **Beta** widens the mainlobe and decreases the amplitude of the window sidelobes in the window's frequency magnitude response. Tunable.

Window sampling

The window sampling, symmetric or periodic. Tunable.

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, N_{fft}, on which to perform the FFT. If N_{fft} exceeds the input frame size, the frame is zero-padded as needed. This parameter is enabled when **Inherit FFT length from input dimensions** is not selected.

Number of spectral averages

The number of spectra to average; setting this parameter to 1 effectively 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

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

To learn how to convert to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

Burg Method	DSP Blockset
Magnitude FFT	DSP Blockset
Window Function	DSP Blockset
Spectrum Scope	DSP Blockset
Yule-Walker Method	DSP Blockset
`pwelch`	Signal Processing Toolbox

See "Power Spectrum Estimation" on page 6-6 for related information. Also see a list of all blocks in the Power Spectrum Estimation library.

Purpose Import a signal from the MATLAB workspace.

Library DSP 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. The frame size is specified by 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. The output sample period is specified by 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 (i.e., 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:

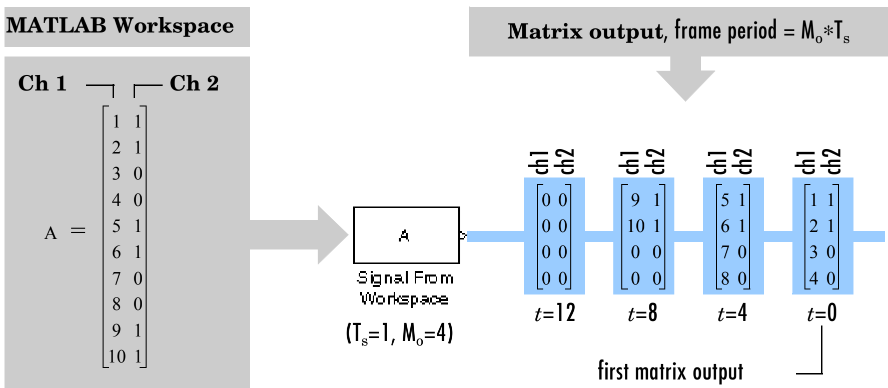
- If **Setting To Zero** is specified, the block generates zero-valued outputs for the duration of the simulation after generating the last frame of the signal.
- If **Holding Final Value** is specified, the block repeats the final sample for the duration of the simulation after generating the last frame of the signal.

- If **Cyclic Repetition** is specified, the block repeats the signal from the beginning after it reaches the last sample in the signal.

Examples

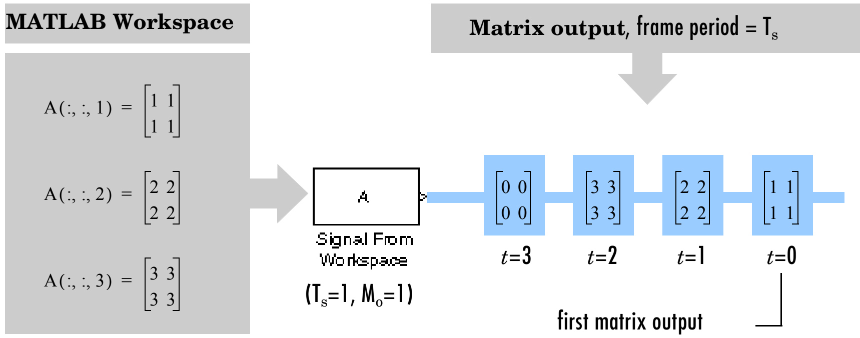
Example 1

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.



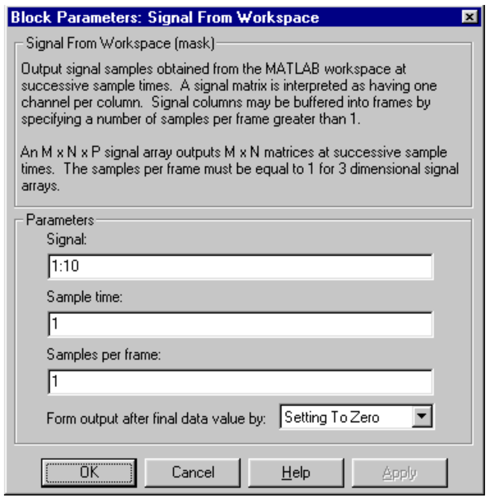
Example 2

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.

Dialog Box


Block Parameters: Signal From Workspace

Signal From Workspace (mask)

Output signal samples obtained from the MATLAB workspace at successive sample times. A signal matrix is interpreted as having one channel per column. Signal columns may be buffered into frames by specifying a number of samples per frame greater than 1.

An M x N x P signal array outputs M x N matrices at successive sample times. The samples per frame must be equal to 1 for 3 dimensional signal arrays.

Parameters

Signal:

1:10

Sample time:

1

Samples per frame:

1

Form output after final data value by: Setting To Zero

OK Cancel Help Apply

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 if a 3-D array is specified 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**). Tunable.

Supported Data Types

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

To learn how to convert to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

From Wave Device	DSP Blockset
From Wave File	DSP Blockset
Signal From Workspace	DSP Blockset
To Workspace	Simulink
Triggered Signal From Workspace	DSP Blockset

See the sections below for related information:

- "Discrete-Time Signals" on page 3-3
- "Multichannel Signals" on page 3-11
- "Benefits of Frame-Based Processing" on page 3-14
- "Creating Signals Using the Signal From Workspace Block" on page 3-38
- "Importing Signals" on page 3-62
- "DSP Sources" on page 7-3 — List of all blocks in the DSP Sources library

Purpose

Write simulation data to an array in the MATLAB main workspace.

Library

DSP 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 if 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 7-521
- "Output Dimension Summary" on page 7-522
- "Matching the Outputs of Signal To Workspace and To Workspace Blocks" on page 7-523
- "Examples" on page 7-524

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. If 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. If the block input is frame-based, this parameter indicates how many *frames of data* to save. If 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`.

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 7-525.)
- **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 7-525.)

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 7-524.)

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.

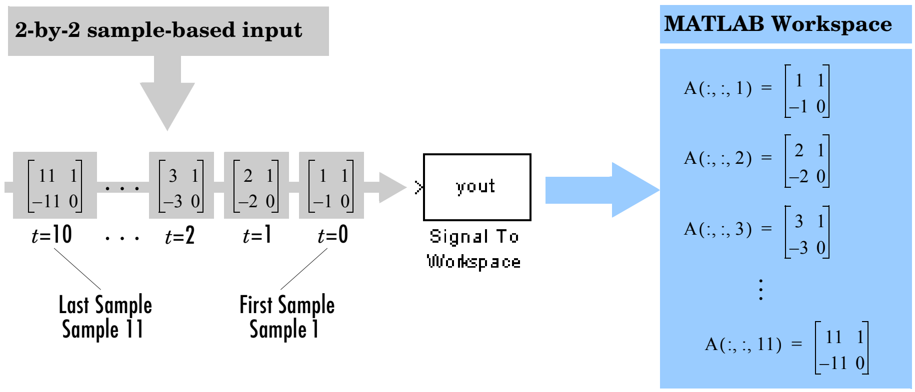
Matching the Outputs of Signal To Workspace and To Workspace Blocks

The To Workspace block in the Simulink Sinks Library and the Signal To Workspace block can output the same array if they are given the same inputs. To match the blocks' outputs, set their parameters as follows.

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 Example 1 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 Example 1 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

Simulation Parameters Dialog Parameters

Start time	`0`
Stop time	`10`

Signal From Workspace Parameters (provides Signal To Workspace input)

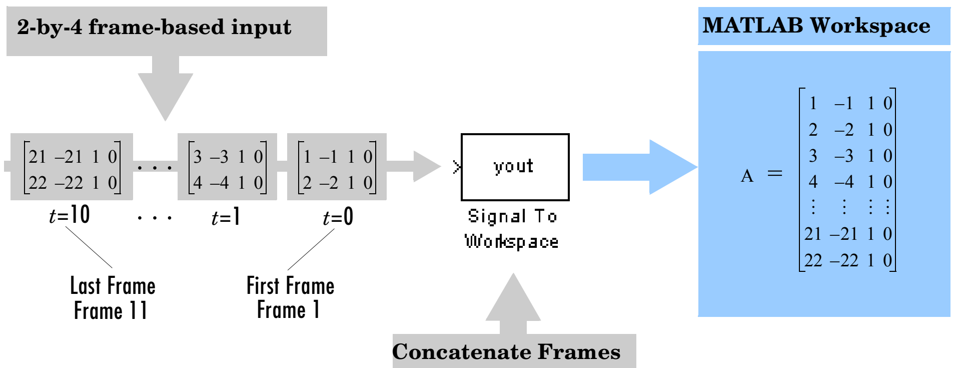
Signal	`input1` (defined below)
Sample time	`1`
Samples per frame	`1`
Form output after final data value by	**Setting to zero**

```
input1 = cat(3, [1 1; -1 0], [2 1; -2 0],...,[11 1; -11 0])
```

Example 2: Frame-Based Inputs. In the following Example 2 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.

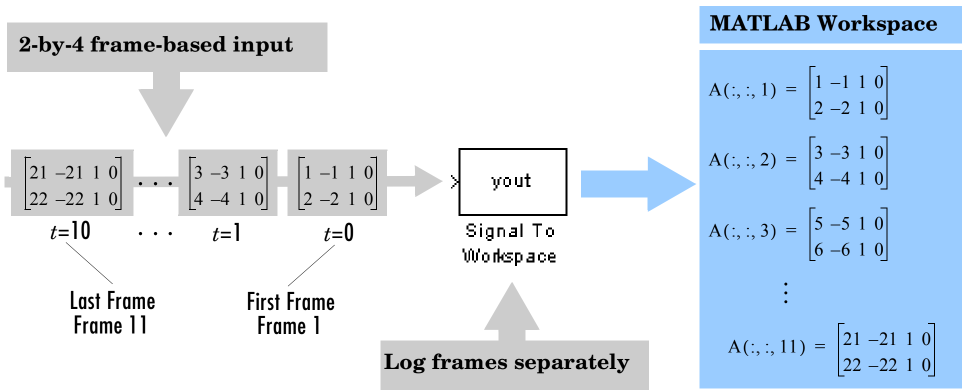
The block settings for the following Example 2 model are similar to the Example 1 block settings, except **Frames** is set to **Concatenate frames (2-D array)** and the Signal From Workspace parameter, **Signal**, is set to `input2`, where

```
input2 = [1 -1 1 0; 2 -2 1 0; 3 -3 1 0;...; 22 -22 1 0]
```

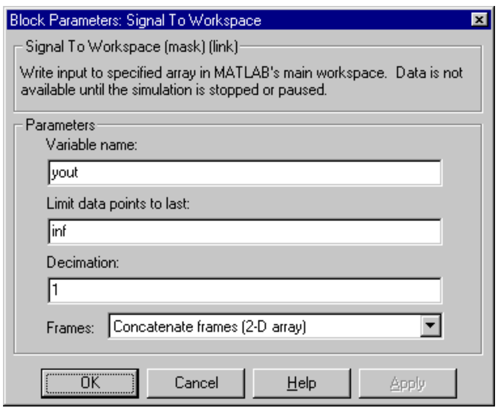


In the 2-D output, there is no indication of where one frame ends and another begins. By setting **Frames** to **Log frames separately (3-D array)** in the Example 2 model, you can easily see each frame in the MATLAB workspace, as illustrated in the following model. Each of the 11 frames is logged separately to create a 2-by-4-by-11 array, A, in the MATLAB workspace.

Signal To Workspace



Dialog Box



Variable name

The name of the array that holds the input data. Tunable.

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. Tunable.

Decimation

The decimation factor, d. Data is written at every dth sample. Tunable.

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. Tunable.

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 to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Triggered To Workspace	DSP Blockset
To Workspace	Simulink

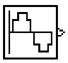
Also see “DSP Sinks” on page 7-3 for a list of all the blocks in the DSP Sinks library.

Sine Wave

Purpose Generate a continuous or discrete sine wave.

Library DSP 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

$$y = A\sin(2\pi f t + \phi)$$

where A is specified by the **Amplitude** parameter, f is specified in hertz by the **Frequency** parameter, and ϕ is specified in radians by the **Phase** 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

$$y = Ae^{j(2\pi f t + \phi)} = A\{\cos(2\pi f t + \phi) + j\sin(2\pi f t + \phi)\}$$

Sections of This Reference Page

Generating Multi-Channel Outputs

For both real and complex sinusoids, the **Amplitude**, **Frequency**, and **Phase** parameter values (A, f, and ϕ) can be scalars or length-N vectors, where N is the desired number of channels in the output. If at least one of these parameters is specified 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** = `[0 0 pi/2]`

$$y = \begin{cases} \sin(2000\pi t) & \text{(channel 1)} \\ 2\sin(1000\pi t) & \text{(channel 2)} \\ 3\sin\left(500\pi t + \frac{\pi}{2}\right) & \text{(channel 3)} \end{cases}$$

Output Sample Time and Samples Per Frame

In all discrete modes (see below), the block buffers the sampled sinusoids into frames of size M, where M is specified by the **Samples per frame** parameter. The output is a frame-based M-by-N matrix with frame period $M*T_s$, where T_s is specified by 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 *i*th 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 DSP 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 look-up, or by a differential method. The three options are explained below.

Discrete Computational Methods

When **Discrete** is selected 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 T_s is specified by the **Sample time** parameter. This mode of operation shares the same benefits and liabilities as the **Continuous** sample mode described above.

If the period of every sinusoid in the output is evenly divisible by the sample period, meaning that $1/(f_i \mathrm{T_s}) = k_i$ is an integer for every output y_i, then the sinusoidal output in the ith channel is a repeating sequence with a period of k_i samples. 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 look-up method at the expense of many more floating-point operations.

Table Lookup. The table look-up 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 if 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, $1/(f_i \mathrm{T_s}) = k_i$ 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 look-up method requires far fewer floating-point operations than any of the other methods, but may 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 (differential) algorithm rather than one based on absolute time. The 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.

$$\begin{aligned}
\sin(t + T_s) &= \sin(t)\cos(T_s) + \cos(t)\sin(T_s) \\
\cos(t + T_s) &= \cos(t)\cos(T_s) - \sin(t)\sin(T_s)
\end{aligned}$$

The update equations for the sinusoid in the ith channel, y_i, can therefore be written in matrix form (for real output) as

$$\begin{bmatrix} \sin\{2\pi f_i(t+T_s)+\phi_i\} \\ \cos\{2\pi f_i(t+T_s)+\phi_i\} \end{bmatrix} = \begin{bmatrix} \cos(2\pi f_i T_s) & \sin(2\pi f_i T_s) \\ -\sin(2\pi f_i T_s) & \cos(2\pi f_i T_s) \end{bmatrix} \begin{bmatrix} \sin(2\pi f_i t+\phi_i) \\ \cos(2\pi f_i t+\phi_i) \end{bmatrix}$$

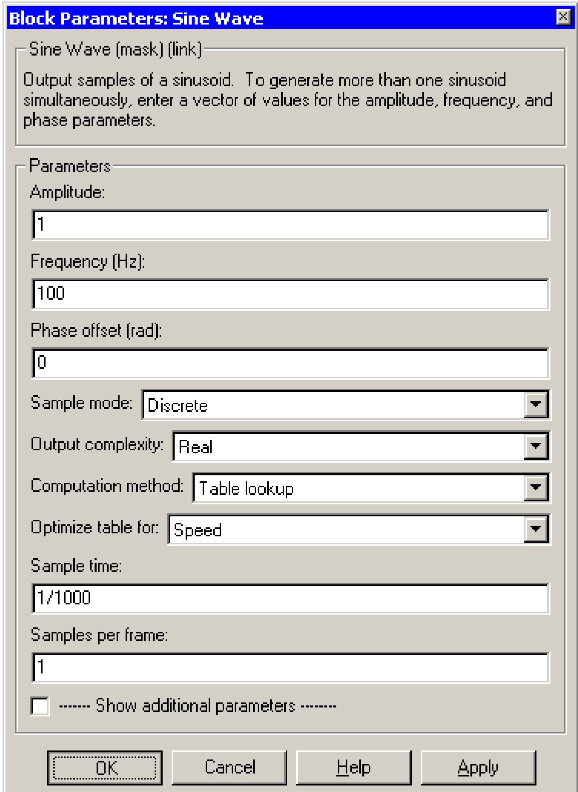
where $\mathrm{T_s}$ is specified by the **Sample time** parameter. Since $\mathrm{T_s}$ is constant, the right-hand matrix is a constant and can be computed once at the start of the simulation. The value of $A_i\sin[2\pi f_i(t+\mathrm{T_s})+\phi_i]$ is then computed from the values of $\sin(2\pi f_i t+\phi_i)$ 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.

Dialog Box



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** parameters. Tunable (when **Computation method** is *not* set to **Table lookup**); when 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** 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. 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. 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. Not tunable in the Simulink external mode when using the differential method.

Sample mode

The block's sampling behavior, **Continuous** or **Discrete**.

Output complexity

The type of waveform to generate: **Real** specifies a real sine wave, **Complex** specifies a complex exponential. Tunable.

Computation method

The method by which discrete-time sinusoids are generated: **Trigonometric fcn**, **Table lookup**, or **Differential**. This parameter is disabled when **Continuous** is selected from the **Sample mode** parameter. For details, see “Discrete Computational Methods” on page 7-530.

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

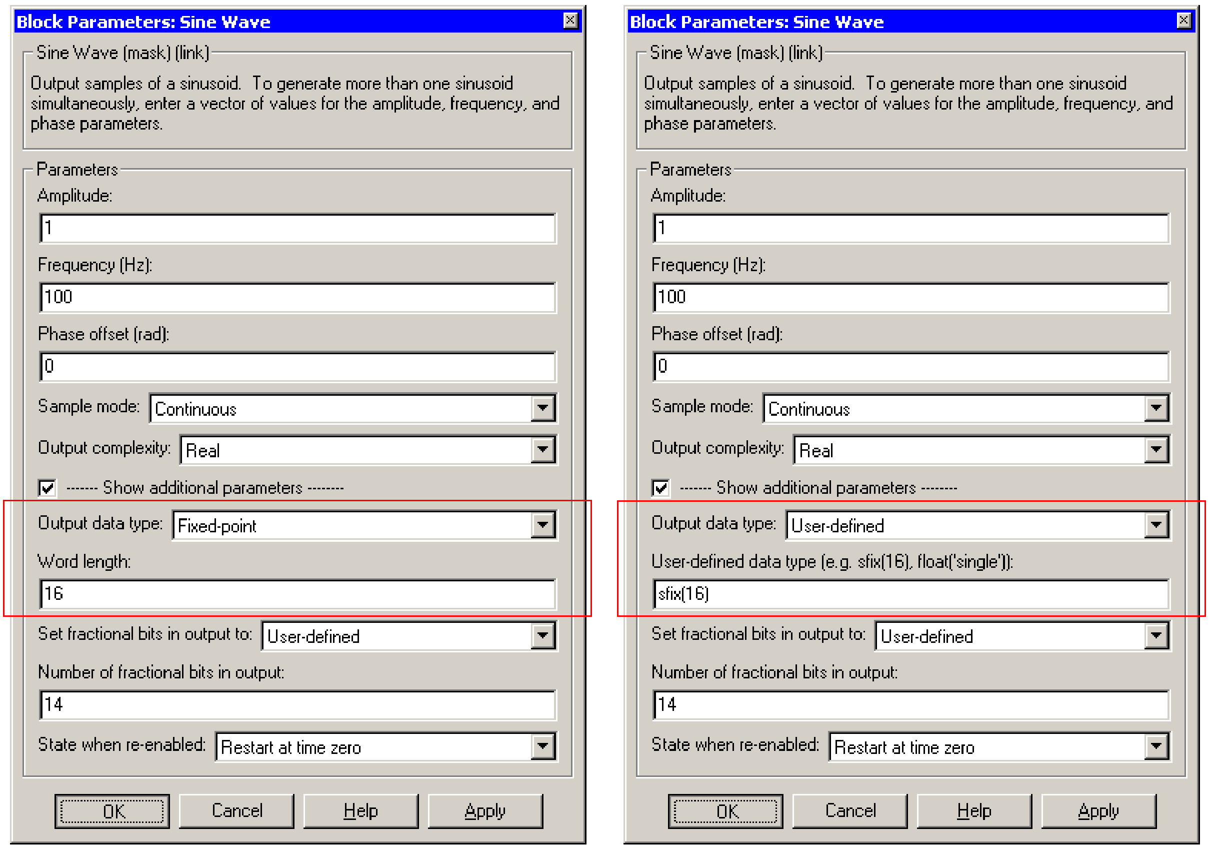
The period with which the sine wave is sampled, T_s. The block's output frame period is $M*T_s$, where M is specified by the **Samples per frame** parameter. This parameter is disabled when **Continuous** is selected from the **Sample mode** parameter.

Samples per frame

The number of consecutive samples from each sinusoid to buffer into the output frame, M. This parameter is disabled when **Continuous** is selected from the **Sample mode** parameter.

Show additional parameters

If selected, additional parameters specific to implementation of the block become visible as shown.



Output data type

Specify the output data type in out of the following ways:

- Choose one of the built-in data types from the drop-down list.

- Choose **Fixed-point** to specify the output data type and scaling in the **Word length**, **Set fractional bits in output to**, and **Number of fractional bits in output** parameters.
- Choose **User-defined** to specify the output data type and scaling in the **User-defined data type**, **Set fractional bits in output to**, and **Number of fractional bits in output** 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 if **Fixed-point** is selected 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 the Fixed-Point Blockset. This parameter is only visible if **User-defined** is selected for the **Output data type** parameter.

Set fractional bits 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 **Number of fractional bits in output** parameter.

This parameter is only visible if **Fixed-point** or **User-defined** is selected for the **Output data type** parameter, and if the specified output data type is a fixed-point data type.

Number of fractional bits in output

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 if **Fixed-point** or **User-defined** is selected for the **Output data type** parameter, and if **User-defined** is selected for the **Set fractional bits in output to** parameter.

State when re-enabled

The behavior of the block when a disabled subsystem containing it is reenabled. 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 **Continuous** is selected from the **Sample mode** parameter.

Supported Data Types

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

To learn how to convert to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

Chirp	DSP Blockset
Complex Exponential	DSP Blockset
Signal From Workspace	DSP Blockset
Signal Generator	Simulink
Sine Wave	Simulink
`sin`	MATLAB

Also see the following topics:

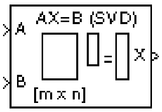
- "Creating Signals Using Signal Generator Blocks" on page 3-36 — How to use this and other blocks to generate signals
- "DSP Sources" on page 7-3 — List of all blocks in the DSP Sources library

Singular Value Decomposition

Purpose Factor a 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.

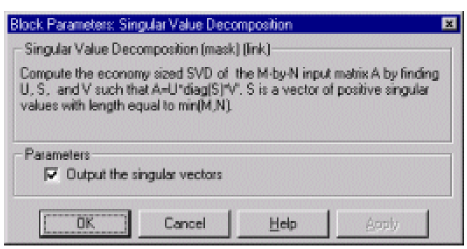
Length-N row inputs are treated as length-N columns.

```
[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



Compute 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 to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

Autocorrelation LPC	DSP Blockset
Cholesky Factorization	DSP Blockset
LDL Factorization	DSP Blockset
LU Inverse	DSP Blockset
Pseudoinverse	DSP Blockset
QR Factorization	DSP Blockset
SVD Solver	DSP Blockset
`svd`	MATLAB

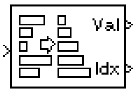
See "Factoring Matrices" on page 6-8 for related information. Also see "Matrix Factorizations" on page 7-10 for a list of all the blocks in the Matrix Factorizations library.

Sort

Purpose Sort the elements in the input by value.

Library Statistics

Description



The Sort block sorts the elements in each column of the input using a *Quicksort* algorithm. The **Mode** parameter specifies the block's mode of operation, and can be set to **Value**, **Index**, or **Value and Index**.

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)             % Equivalent MATLAB code (ascending)
val = flipud(sort(u))     % Equivalent MATLAB code (descending)
```

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 M-by-N matrix containing the sorted columns of `u`. Complex inputs are sorted by magnitude, and the output has the same frame status as the input.

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)        % Equivalent MATLAB code (ascending)
[val,idx] = flipud(sort(u))% Equivalent MATLAB code (descending)
```

and outputs the sample-based M-by-N index matrix, `idx`. The *j*th column of `idx` is an index vector that permutes the *j*th column of `u` to the desired sorting order:

```
val(:,j) = u(idx(:,j),j)
```

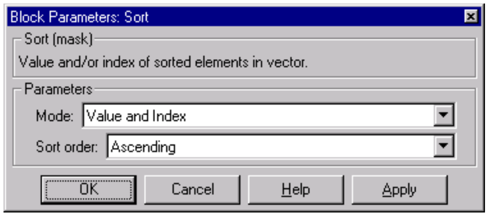
The index value outputs are always 32-bit unsigned integer values.

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`.

Dialog Box



Mode

The block's mode of operation: Output the sorted matrix (**Value**), the index matrix (**Index**), or both (**Value and Index**).

Sort order

The order in which to sort the input values, **Descending** or **Ascending**. Tunable, except in the Simulink external mode.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- 32-bit unsigned integer — The optional `Idx` port always outputs 32-bit unsigned integer values.

To learn how to convert to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

Histogram	DSP Blockset
Median	DSP Blockset
`sort`	MATLAB

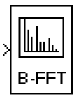
Also see "Statistics" on page 7-18 for a list of all the blocks in the Statistics library.

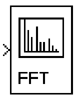
Spectrum Scope

Purpose Compute and display the short-time FFT of each input signal.

Library DSP Sinks

Description





The Spectrum Scope block computes and displays the magnitude-squared FFT of the input. The input be a 1-D vector or a 2-D matrix of any frame status.

When the 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 checking the **Buffer input** check box and specifying the new buffer size. In the latter case, you can also specify an optional **Buffer overlap**.

Buffering 1-D vector inputs is recommended. In this case, the inputs are buffered 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 a 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 be able to see the plot, and a warning is also displayed on the scope itself. It is not recommended that you leave 1-D inputs unbuffered.

The number of input samples that the block buffers before computing and displaying the magnitude FFT is specified by the **Buffer size** parameter, M_o. The **Buffer overlap** parameter, L, specifies 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 $(M_o\text{-}L)*T_s$, where T_s is the input sample period, and is *equal* to the input sample period when the **Buffer overlap** is M_o-1. For negative **Buffer overlap** values, the block simply 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.

When the **FFT length** check box is cleared and the input is buffered, the block uses the buffer size as the FFT size. If the check box is cleared and the input is not buffered, the block uses the input size as the FFT size. When the check box is selected, the **FFT length** parameter, N_{fft}, is enabled, and specifies the number of samples on which 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 Short-Time FFT for more information.

In order to correctly scale the frequency axis (i.e., to determine the frequencies against which the transformed input data should be plotted), the block needs to know the actual sample period of the time-domain input. This is specified by the **Sample time of original time series** parameter, T_s.

When the **Inherit sample time from input** check box is selected, 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.

One example when these conditions do not hold, is such as when the input to the block is not the original signal, but a zero-padded or otherwise rate-altered version. In such cases, you should specify the appropriate value for the **Sample time of original time-series** parameter.

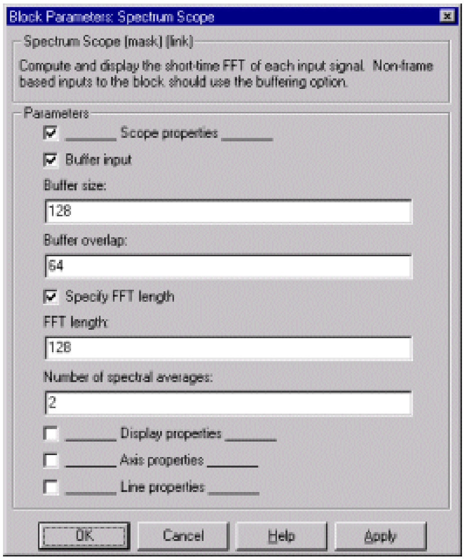
The **Frequency units** parameter specifies whether the frequency axis values should be in units of Hertz or rad/s, and the **Frequency range** parameter specifies the range of frequencies over which the magnitudes in the input should be plotted. The available options are **[0..Fs/2]**, **[-Fs/2..Fs/2]**, and **[0..Fs]**, where F_s is the time-domain signal's actual sample frequency. If 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 $2\pi/(N_{fft}T_s)$.

Note that all of the FFT-based blocks in the DSP 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.

Spectrum Scope

For information about the scope window, as well as the **Display properties**, **Axis properties**, and **Line properties** panels in the dialog box, see the reference page for the Vector Scope block.

Dialog Box



Scope properties

Select to expose **Scope properties** panel. Tunable.

Buffer input

Select to expose **Buffer input** panel. Tunable.

Buffer size

The number of signal samples to include in each buffer. Tunable.

Buffer overlap

The number of samples by which consecutive buffers overlap. Tunable.

Specify FFT length

Select to expose **Specify FFT length** panel. Tunable.

FFT length

The number of samples on which to perform the FFT. If the **FFT length** differs from the buffer size, the data is zero-padded or truncated as needed. Tunable.

Number of spectral averages

The number of spectra to average. Setting this parameter to 1 effectively disables averaging. See Short-Time FFT for more information. Tunable.

Display properties

Select to expose the **Display properties** panel. See Vector Scope for more information. Tunable.

Axis properties

Select to expose the **Axis properties** panel. See Vector Scope for more information. Tunable.

Line properties

Select to expose the **Line properties** panel. See Vector Scope for more information. Tunable.

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 to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

FFT	DSP Blockset
Vector Scope	DSP Blockset

Also see the following topics:

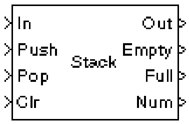
- "Viewing Signals" on page 3-80 — How to use this and other blocks to view signals
- "DSP Sinks" on page 7-3 — List of all blocks in the DSP Sinks library

Stack

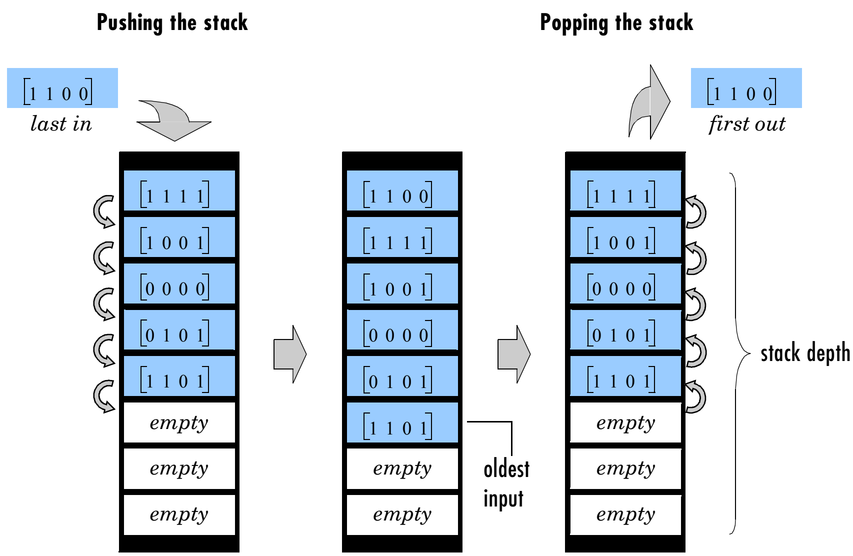
Purpose Store inputs into a LIFO register.

Library Signal Management / Buffers

Description The Stack block stores a sequence of input samples in a LIFO (last in, first out) 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. If **Clear output port on reset** is selected, 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 re-enabled; the `Out` port value is only reset to zero in this case if **Clear output port on reset** is selected.

When two or more of the control input ports are triggered at the same time step, the operations are executed in the following order:

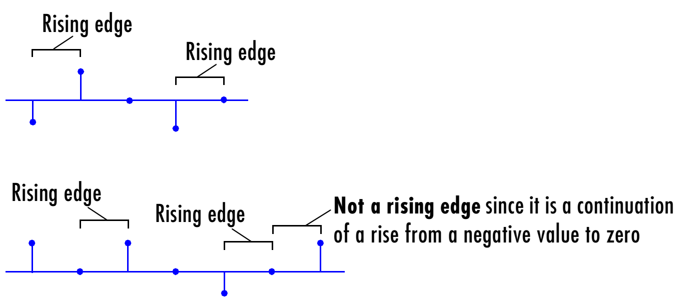
1 `Clr`

2 `Push`

3 `Pop`

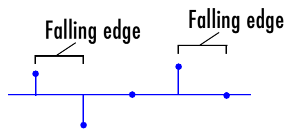
The rate of the trigger signal must be a positive integer multiple of the rate of the data signal input. The triggering event for the `Push`, `Pop`, and `Clr` ports is specified by the **Trigger type** pop-up menu, and can be one of the following:

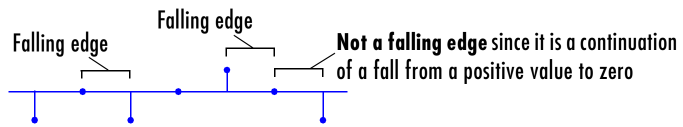
- **Rising edge** — Triggers execution of the block when the trigger input does one of the following:
  - Rises from a negative value to a positive value or zero
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)



- **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 3-91 and the topic on the **Simulation Parameters** dialog box in the Simulink 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.

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
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

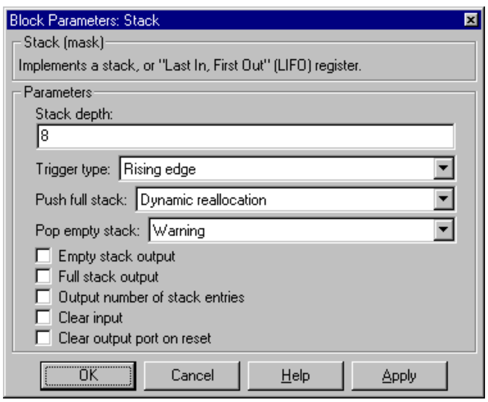
In	Push	Pop	Clr	Stack	Out	Empty	Full	Num
10	1	0	0	*top* 10 *bottom*	2	0	0	1
11	0	0	0	*top* 11 10 *bottom*	2	0	0	2
12	1	0	1	*top* 12 *bottom*	0	0	0	1

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



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 a positive integer multiple of the rate of the data signal input. Tunable only in simulation (not tunable in Real-Time Workshop external mode or in the Simulink Performance Tools Accelerator).

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 **Dynamic reallocation** is selected 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

- Double-precision floating point
- Single-precision floating point
- Fixed-point
- Custom data types

- Boolean — The block accepts Boolean inputs to the `Push`, `Pop`, and `Clr` ports. The block may output Boolean values at the `Out` and `Full` ports 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 A-11. To learn how to disable Boolean output support, see “Steps to Disabling Boolean Support” on page A-12.
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

To learn how to convert to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

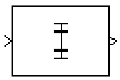
Buffer	DSP Blockset
Delay Line	DSP Blockset
Queue	DSP Blockset

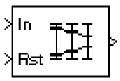
Also see “Buffers” on page 7-14 for a list of all the blocks in the Buffers library.

Purpose Find the 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 the **Running standard deviation** check box is *not* selected, 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.

$$\sigma_j = \sqrt{\sigma_{j,Re}^2 + \sigma_{j,Im}^2}$$

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

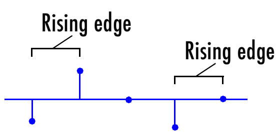
When the **Running standard deviation** check box is selected, the block tracks the standard deviation of each channel in a *time-sequence* of M-by-N inputs. For sample-based inputs, the output is a sample-based M-by-N matrix with each element y_{ij} containing the standard deviation of element u_{ij} 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 jth column over all inputs since the last reset, up to and including element u_{ij} of the current input.

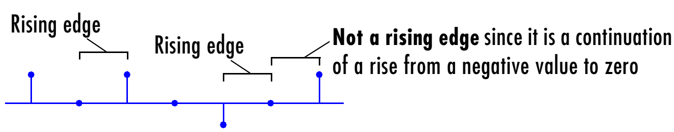
As in basic operation, length-M 1-D vector inputs and *sample-based* length-M row vector inputs are both treated as M-by-1 column vectors.

Resetting the Running 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.

The reset event is specified by the **Reset port** parameter, and can be one of the following:

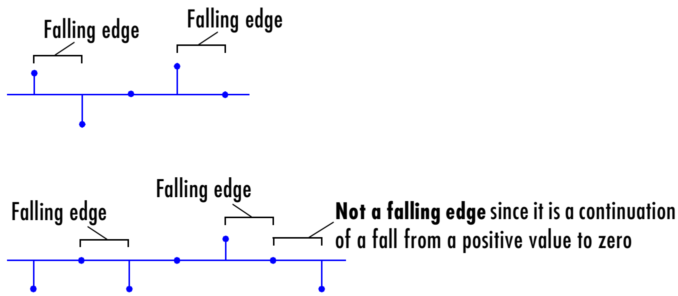
- **None** disables the `Rst` port.
- **Rising edge** — Triggers a reset operation when the `Rst` input does one of the following:
  - Rises from a negative value to a positive value or zero
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value 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)

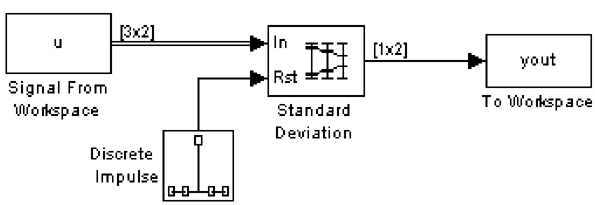


- **Either edge** — Triggers a reset operation when the `Rst` input is a **Rising edge** or **Falling edge** (as described above).
- **Non-zero sample** — Triggers a reset operation at each sample time that the `Rst` input is not zero.

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 3-91 and the topic on the **Simulation Parameters** dialog box in the Simulink documentation.

Example

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

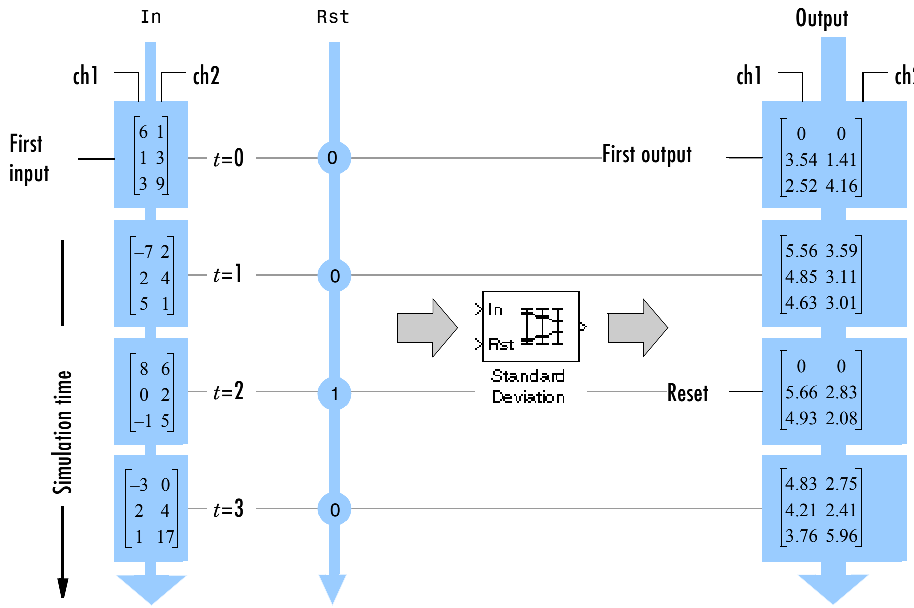
where

```
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]'
```

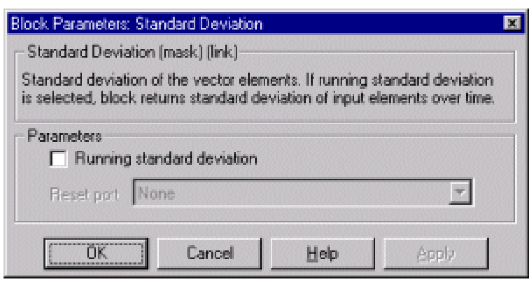
The Discrete Impulse block has the following settings:

- **Delay (samples)** = 2
- **Sample time** = 1
- **Samples per frame** = 1

The block's operation is shown in the figure below.



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 set the **Running standard deviation** parameter. For more information, see “Resetting the Running Standard Deviation” on page 7-554.

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 to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Mean	DSP Blockset
RMS	DSP Blockset
Variance	DSP Blockset
`std`	MATLAB

Also see “Statistics” on page 7-18 for a list of all the blocks in the Statistics library.

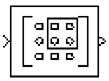
Purpose

Select a subset of elements (submatrix) from a 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 **One row** or **Range of rows** is selected from the **Row span** parameter, the desired row or range of rows is specified by the **Row** parameter, or the **Starting row** and **Ending row** parameters. Similarly, when **One column** or **Range of columns** is selected from the **Column span** parameter, the desired column or range of columns is specified by 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`. If 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`. If 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`. If 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`. If 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`. If 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`. If all rows are to be included, this is equivalent to `y(:,1) = u(:,N-firstcol)`.

- **Last**

  For rows, this specifies that the last row of `u` should be used as the only row of `y`. If 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`. If 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`. If 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`. If 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`. If 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`. If 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`. If 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`. If 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`. If 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`. If 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`. If 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`. If 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`. If 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`. If all rows are to be included, this is equivalent to `y(:,end) = u(:,N/2-lastcol)`.

- **Middle**

  For rows, this specifies that the middle row of `u` should be used as the last row of `y`. If all columns are to be included, this is equivalent to `y(end,:) = u(M/2,:)`.

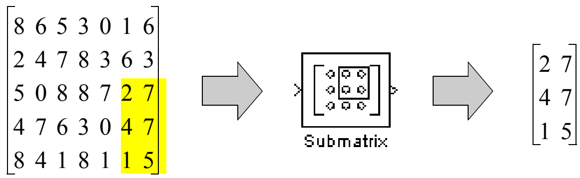
  For columns, this specifies that the middle column of `u` should be used as the last column of `y`. If all rows are to be included, this is equivalent to `y(:,end) = u(:,N/2)`.

Example

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**

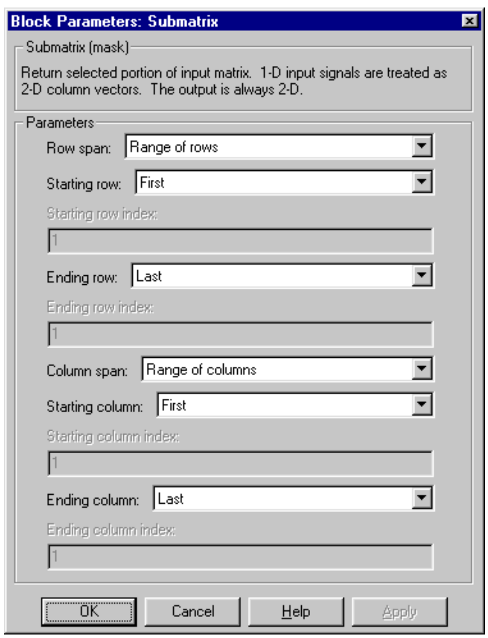
The figure below shows the operation for a 5-by-7 matrix with random integer elements, `randint(5,7,10)`.



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**.

Row/Starting row

The input row to be used as the first row of the output. **Row** is enabled when **One row** is selected from **Row span**, and **Starting row** when **Range of rows** is selected 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 **Index** is selected from Row, and **Starting row index** when **Index** is selected 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 **Offset from middle** or **Offset from last** is selected from **Row**, and Starting row offset is enabled when **Offset from middle** or **Offset from last** is selected from **Starting row**.

Ending row

The input row to be used as the last row of the output. This parameter is enabled when **Range of rows** is selected from **Row span** and any option but **Last** is selected 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 **Index** is selected 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 **Offset from middle** or **Offset from last** is selected 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 **One column** is selected from **Column span**, and **Starting column** is enabled when **Range of columns** is selected 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 **Index** is selected from Column, and **Starting column index** is enabled when **Index** is selected 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 **Offset from middle** or **Offset from last** is selected from Column. **Starting column offset** is enabled when **Offset from middle** or **Offset from last** is selected from **Starting column**.

Ending column

The input column to be used as the last column of the output. This parameter is enabled when **Range of columns** is selected from **Column span** and any option but **Last** is selected 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 **Index** is selected 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 **Offset from middle** or **Offset from last** is selected from **Ending column**.

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 to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Reshape	Simulink
Selector	Simulink
Variable Selector	DSP Blockset
`reshape`	MATLAB

See “Deconstructing Signals” on page 3-54 for related information. Also see “Matrix Operations” on page 7-11 and “Indexing” on page 7-14 for a list of all the blocks in these libraries.

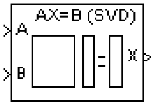
Purpose

Solve the equation AX=B using singular value decomposition.

Library

Math Functions / Matrices and Linear Algebra / Linear System Solvers

Description

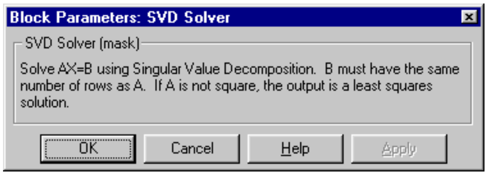


The SVD Solver block solves the linear system AX=B, which can be overdetermined, underdetermined, or exactly determined. The system is solved by applying SVD factorization to the M-by-N matrix, A, at the `A` port. The input to the `B` port is the right hand-side M-by-L matrix, B. A length-M 1-D vector 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 is applied to an underdetermined system, the output X is chosen such that the number of nonzero entries in X is minimized.

Dialog Box



Supported Data Types

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

To learn how to convert to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Autocorrelation LPC	DSP Blockset
Cholesky Solver	DSP Blockset
LDL Solver	DSP Blockset
Levinson-Durbin	DSP Blockset
LU Inverse	DSP Blockset
Pseudoinverse	DSP Blockset
QR Solver	DSP Blockset
Singular Value Decomposition	DSP Blockset

See “Solving Linear Systems” on page 6-7 for related information. Also see “Linear System Solvers” on page 7-9 for a list of all the blocks in the Linear System Solvers library.

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.

Library

DSP Sinks

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 to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

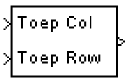
See "DSP Sinks" on page 7-3 for a list of all the blocks in the DSP Sinks library.

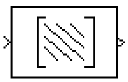
Toeplitz

Purpose Generate a 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

$$\begin{bmatrix} 1 & 7 & 3 & 3 & 2 & 1 & 3 \\ 2 & 1 & 7 & 3 & 3 & 2 & 1 \\ 3 & 2 & 1 & 7 & 3 & 3 & 2 \\ 4 & 3 & 2 & 1 & 7 & 3 & 3 \\ 5 & 4 & 3 & 2 & 1 & 7 & 3 \end{bmatrix}$$

If both of the inputs are sample-based, the output is sample-based. Otherwise, the output is frame-based.

When the **Symmetric** check box is selected, the block generates a symmetric (Hermitian) Toeplitz matrix from a single input, `u`, defining both the first row and first column of the matrix.

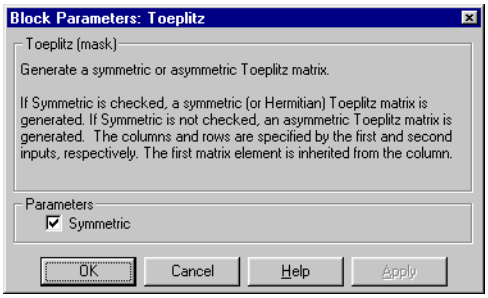
```
y = toeplitz(u)        % Equivalent MATLAB code
```

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.

Dialog Box



Symmetric

When selected, enables the single-input configuration for symmetric Toeplitz matrix output.

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 to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Constant Diagonal Matrix	DSP Blockset
`toeplitz`	MATLAB

Also see “Matrix Operations” on page 7-11 for a list of all the blocks in the Matrix Operations library.

To Wave Device

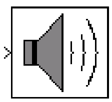
Purpose

Send audio data to a 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 PCM (pulse code modulation) format, and should typically be sampled at one of the standard Windows audio device rates: 8000, 11025, 22050, or 44100 Hz. Some hardware may support other rates in addition to these.

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 target. In cases when the default sound device is *not* the desired output device, clear **Use default audio device**, and select 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. If the input data type is `uint8`, the block conveys the signal samples to the audio device using 8 bits. If 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** parameter.

```
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.

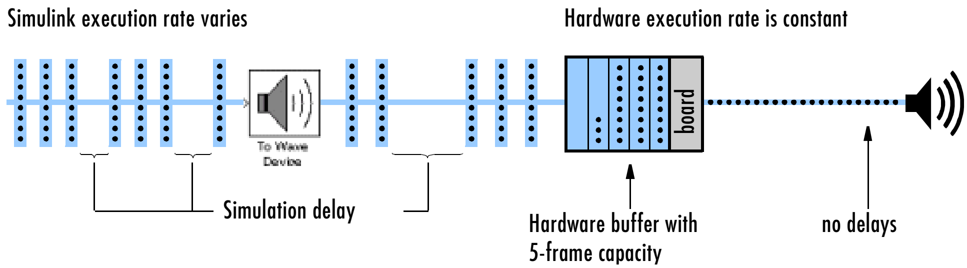
Input Data Type	Valid Input Amplitude Range
double	±1
single	±1
int16	-32768 to 32767 (-2^{15} to $2^{15} - 1$)
uint8	0 to 255

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 T_d is specified by 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. The size of the buffer, T_b, is specified by 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 8 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 8kHz, for instance, this small buffer could hold approximately 0.005 second of data.)



If 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 (i.e., 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, if the buffer size is too small in relation to the simulation throughput rate, the buffer may 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

  $$\frac{T_b F_s}{M_o}$$

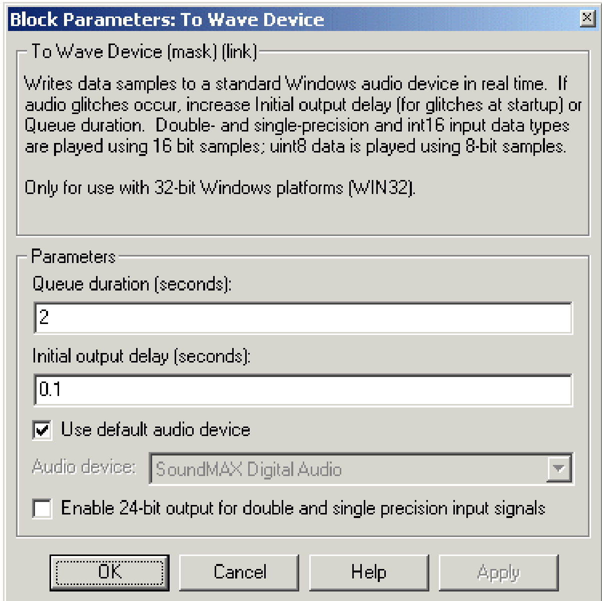
  where F_s is the sample rate of the signal and M_o is the number of samples per frame. The optimal buffer size for a given signal depends on the signal length, the frame size, and the speed of the simulation. The maximum number of frames that can be buffered is 1024.
- *Increase the simulation throughput 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 3-85 for other ideas on improving simulation performance.

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** if 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 data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

From Wave Device	DSP Blockset
To Wave File	DSP Blockset
`audiodevinfo`	MATLAB
`audioplayer`	MATLAB
`sound`	MATLAB

See "Exporting and Playing WAV Files" on page 3-79 for related information. Also see "Windows (WIN32)" on page 7-13 for a list of all the blocks in the Windows (WIN32) library.

To Wave File

Purpose

Write audio data to file in the Microsoft Wave (`.wav`) format (32-bit Windows operating systems only).

Library

Platform-specific I/O / Windows (WIN32)

Description

To Wave

The To Wave File block writes audio data to a Microsoft Wave (`.wav`) file in the uncompressed PCM (pulse code modulation) 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
```

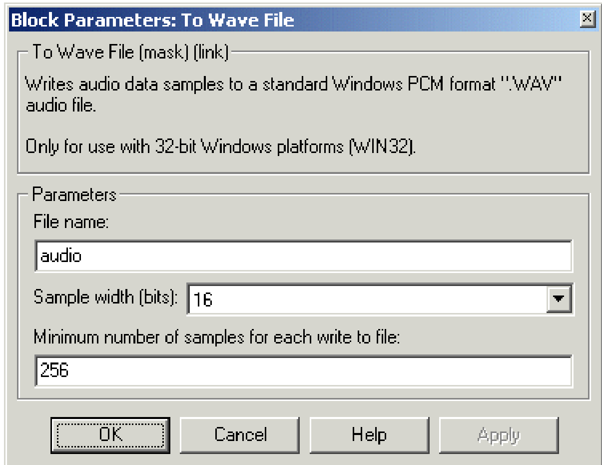
The **Sample Width (bits)** parameter specifies the number of bits used to represent the signal samples in the file. Two settings are available:

- **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 L is specified by 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.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- 16-bit signed integer
- 8-bit unsigned integer

To learn how to convert data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

From Wave File	DSP Blockset
To Wave Device	DSP Blockset
To Workspace	Simulink
`wavwrite`	MATLAB

See “Exporting and Playing WAV Files” on page 3-79 for related information. Also see “Windows (WIN32)” on page 7-13 for a list of all the blocks in the Windows (WIN32) library.

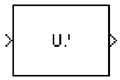
Purpose

Compute the transpose of a matrix.

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 the **Hermitian** check box is selected, the block performs the Hermitian (complex conjugate) transpose

```
y = u'          % Equivalent MATLAB code
```

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \end{bmatrix} \xrightarrow{\texttt{u'}} \begin{bmatrix} u_{11}^{*} & u_{21}^{*} \\ u_{12}^{*} & u_{22}^{*} \\ u_{13}^{*} & u_{23}^{*} \end{bmatrix}$$

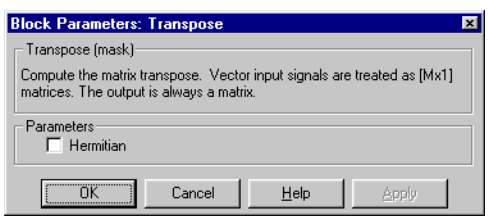
When the **Hermitian** check box is not selected, the block performs the nonconjugate transpose

```
y = u.'         % Equivalent MATLAB code
```

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \end{bmatrix} \xrightarrow{\texttt{u.'}} \begin{bmatrix} u_{11} & u_{21} \\ u_{12} & u_{22} \\ u_{13} & u_{23} \end{bmatrix}$$

A length-M 1-D vector input is treated as an M-by-1 matrix. The output is always sample-based.

Dialog Box



Hermitian

When selected, specifies the complex conjugate transpose. Tunable, except in the Simulink external mode.

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 to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

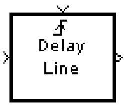
Permute Matrix	DSP Blockset
Reshape	Simulink
Submatrix	DSP Blockset

Also see "Matrix Operations" on page 7-11 for a list of all the blocks in the Matrix Operations library.

Purpose Buffer a sequence of inputs into a 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 M_o is specified by 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 (⨍). The newly acquired input sample is appended to the output frame (when the next triggering event occurs) 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.

The triggering event at input 2 is specified by the **Trigger type** pop-up menu, and can be one of the following:

- **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 M_o is specified by 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.

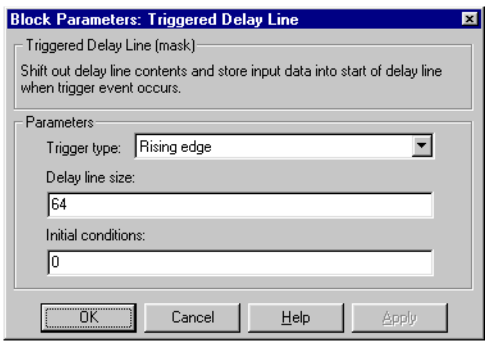
Frame-Based Operation

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 (i.e., 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). If 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. If 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



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

- 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 to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Buffer	DSP Blockset
Delay Line	DSP Blockset
Unbuffer	DSP Blockset

Also see “Buffers” on page 7-14 for a list of all the blocks in the Buffers library.

Triggered Signal From Workspace

Purpose Import signal samples from the MATLAB workspace when triggered.

Library DSP Sources

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 ( ). 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. The frame size is specified by 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

The triggering event at the input port is specified by the **Trigger type** pop-up menu, and can be one of the following:

- **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.

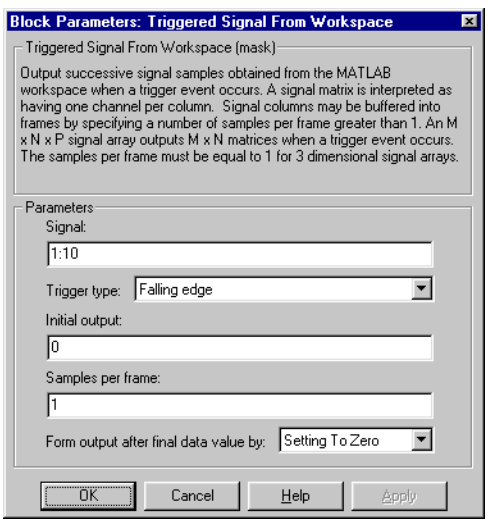
Initial and Final Conditions

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 (i.e., 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:

- If **Setting To Zero** is specified, the block generates zero-valued outputs for the duration of the simulation after generating the last frame of the signal.
- If **Holding Final Value** is specified, the block repeats the final sample for the duration of the simulation after generating the last frame of the signal.
- If **Cyclic Repetition** is specified, the block repeats the signal from the beginning after generating the last frame. If 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 (e.g., 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 if a 3-D array is specified 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**).

Supported Data Types

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

To learn how to convert to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

From Wave Device	DSP Blockset
From Wave File	DSP Blockset
Signal To Workspace	DSP Blockset
Signal From Workspace	DSP Blockset
Triggered To Workspace	DSP Blockset

See the sections below for related information:

- "Discrete-Time Signals" on page 3-3
- "Multichannel Signals" on page 3-11
- "Benefits of Frame-Based Processing" on page 3-14
- "Creating Signals Using the Signal From Workspace Block" on page 3-38
- "Importing Signals" on page 3-62
- "DSP Sources" on page 7-3 — List of all blocks in the DSP Sources library

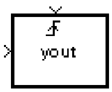
Purpose

Write the input sample to the workspace when triggered.

Library

DSP Sinks

Description



The Triggered To Workspace block creates a matrix or array variable in the workspace, where it stores the acquired inputs at the end of a simulation. An existing variable with the same name is overwritten.

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. (If 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. (If 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 (). At all other times, the block ignores input 1. The triggering event at input 2 is specified by the **Trigger type** pop-up menu, and can be one of the following:

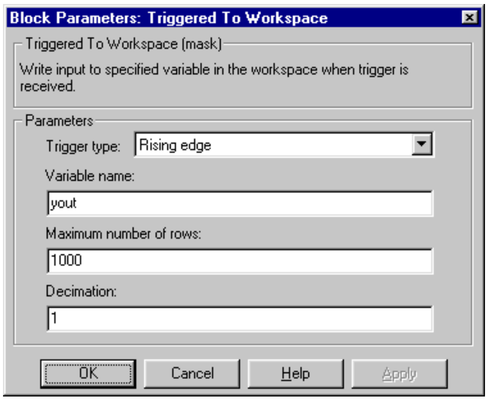
- **Rising edge** triggers execution of the block when the trigger input rises from a negative value to zero or a positive value, or from zero to a positive value.
- **Falling edge** triggers execution of the block when the trigger input falls from a positive value to zero or a negative value, or from zero to a negative value.
- **Either edge** triggers execution of the block when either a rising or falling edge (as described above) occurs.

To save a record of the sample time corresponding to each sample value, check the **Time** box in the **Save to workspace** parameters list of the **Simulation**

Parameters dialog. You can access these parameters by selecting **Parameters** from the **Simulation** menu, and clicking on the **Workspace I/O** tab.

The nontriggered version of this block is To Workspace.

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.

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 to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Signal From Workspace	DSP Blockset
To Workspace	Simulink

See “Exporting Signals” on page 3-72 for related information.

Two-Channel Analysis Subband Filter

Purpose

Decompose a signal into a high-frequency subband and a low-frequency subband

Library

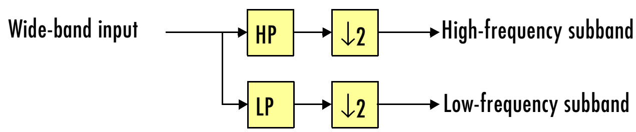
Filtering / Multirate Filters

Description

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 7-596.

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 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.

See other sections of this reference page for more information.

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. If 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 in both this block, and the filters 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 7-599.

The block initializes all filter states to zero.

Sample-Based Operation

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 simulation parameters, some sample-based outputs may have one sample of latency, as described in "Latency" on page 7-595.

Frame-Based Operation

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 7-595.
- **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 *n*th 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 *n*th 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.

For more information about block rates and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" on page 3-91 and the topic on the **Simulation Parameters** dialog box in the Simulink documentation.

Table 7-20: Amount of Block Latency for All Possible Block Settings

Input	Latency	No Latency
Sample-based	One sample of latency when **Mode** = **MultiTasking** or **Auto** in the **Simulation Parameters** dialog (Ctrl + E). First output sample of each channel is always `0`.	**Mode** = **SingleTasking** in the **Simulation Parameters** dialog (Ctrl + E).
Frame-based	One frame of latency when **Framing = Maintain input frame size**. First output frame is always all zeros.	**Framing = Maintain input frame rate**

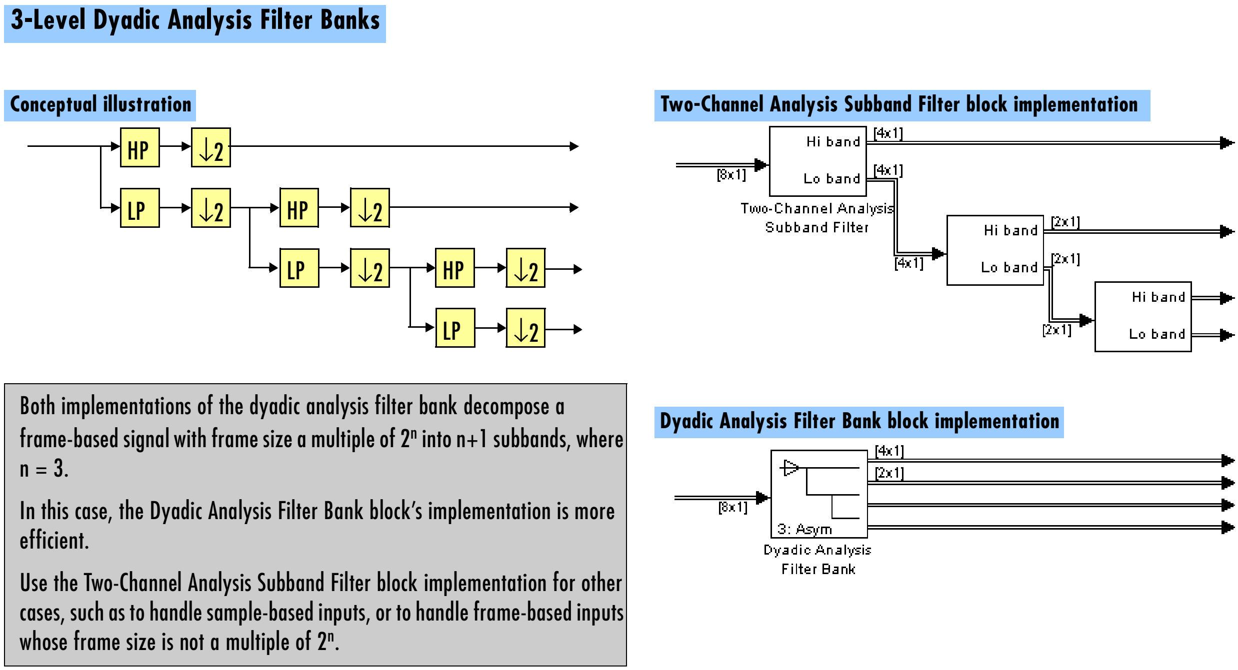
Creating Multilevel Dyadic Analysis Filter Banks

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 "Review of Dyadic Analysis Filter Banks" on page 7-191 in the Dyadic Analysis Filter Bank reference page.)

Note 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 Table 7-20, Amount of Block Latency for All Possible Block Settings, on page 7-595.)

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 more efficient (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.



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.

Examples

See the following DSP Blockset demos, which use the Two-Channel Analysis Subband Filter block:

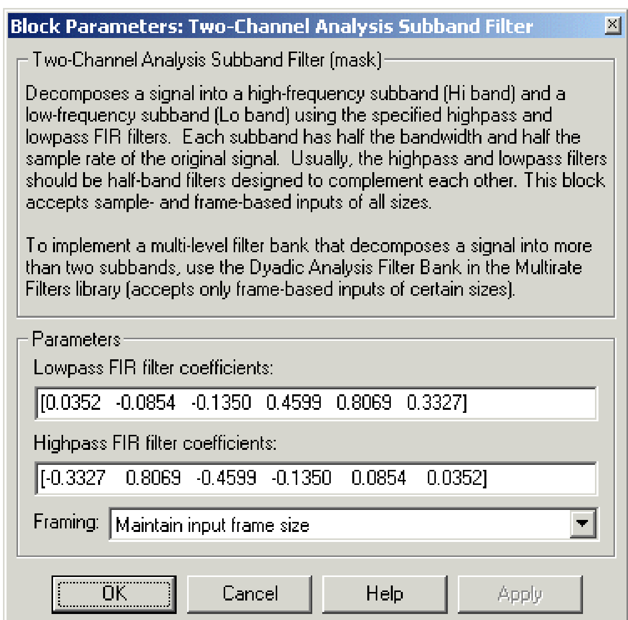
- Multi-level PR filter bank
- Denoising
- Wavelet transmultiplexer (WTM)

Note 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.

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



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. The default values of this parameter specifies a filter based on a 3rd-order Daubechies wavelet. If 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. For more information, see "Specifying the FIR Filters" on page 7-593.

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. The default values of this parameter specifies a filter based on a 3rd-order Daubechies wavelet. If 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. For more information, see "Specifying the FIR Filters" on page 7-593.

Framing

For frame-based inputs, the method by which to implement the decimation: by halving the output frame rate (**Maintain input frame size**), or halving the output frame size (**Maintain input frame rate**). For more information, see "Frame-Based Operation" on page 7-594. Some settings of this parameter causes the block to have nonzero latency, as described in "Latency" on page 7-595.

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.

Two-Channel Analysis Subband Filter

Supported Data Types

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

To learn how to convert to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Dyadic Analysis Filter Bank	DSP Blockset
Two-Channel Synthesis Subband Filter	DSP Blockset
`fir1`	Signal Processing Toolbox
`fir2`	Signal Processing Toolbox
`firls`	Signal Processing Toolbox
`remez`	Signal Processing Toolbox

For related information, see “Multirate Filters” on page 4-32. Also see “Filtering” on page 7-6 for a list of all DSP Blockset filtering blocks.

Two-Channel Synthesis Subband Filter

Purpose

Reconstruct a signal from a high-frequency subband and a low-frequency subband

Library

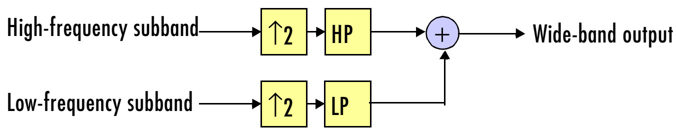
Filtering / Multirate Filters

Description

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 7-605.

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 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.

See other sections of this reference page for more information.

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 7-608.

The block initializes all filter states to zero.

Sample-Based Operation

Valid Sample-Based Inputs. The block accepts any two M-by-N sample-based matrices with the same sample rates. The block treats each matrix as $M \cdot N$ independent subbands, where 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 simulation parameters, some sample-based outputs may have one sample of latency, as described in “Latency” on page 7-604.

Frame-Based Operation

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 7-595.
- **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 *n*th 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 *n*th 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.

For more information about block rates and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" on page 3-91 and the topic on the **Simulation Parameters** dialog box in the Simulink documentation.

Table 7-21: Amount of Block Latency for All Possible Block Settings

Input	Latency	No Latency
Sample-based	One sample of latency when **Mode** = **MultiTasking** or **Auto** in the **Simulation Parameters** dialog (Ctrl + E). First output sample of each channel is always `0`.	**Mode** = **SingleTasking** in the **Simulation Parameters** dialog (Ctrl + E).
Frame-based	One frame of latency when **Framing = Maintain input frame size**. First output frame is always all zeros.	**Framing = Maintain input frame rate**

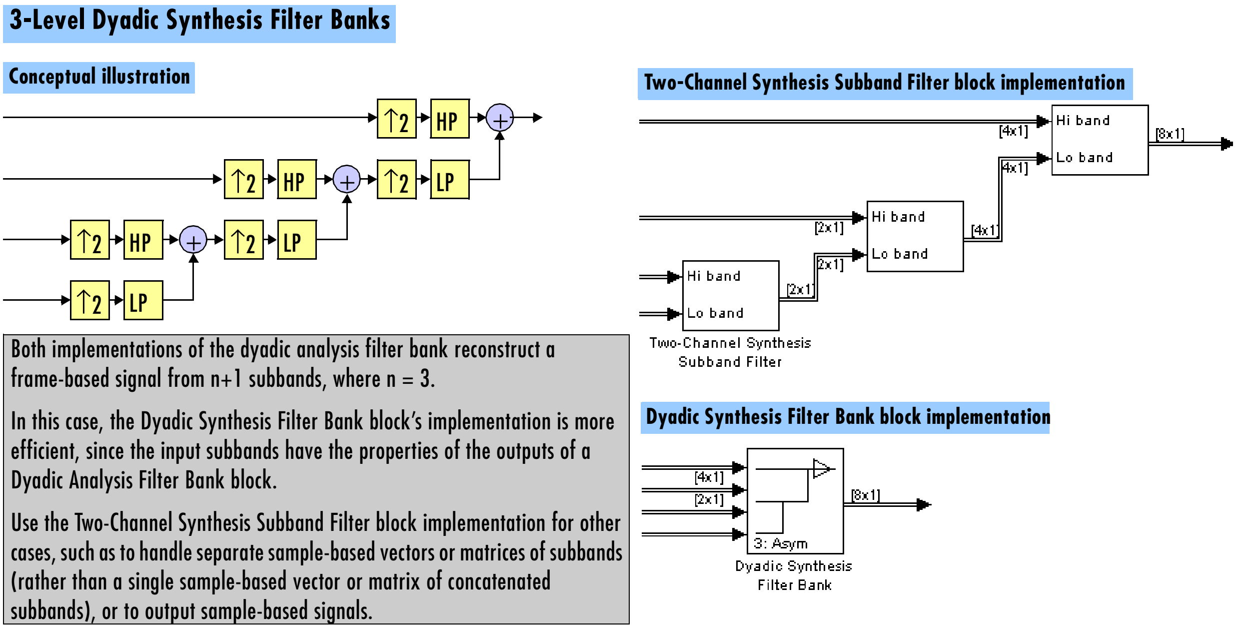
Creating Multilevel Dyadic Synthesis Filter Banks

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 "Review of Dyadic Synthesis Filter Banks" on page 7-205 in the Dyadic Synthesis Filter Bank reference page.)

Note 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 Table 7-21, Amount of Block Latency for All Possible Block Settings, on page 7-604.)

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 more efficient (faster and requires less memory) to use the Dyadic Synthesis Filter Bank block. In particular, use the Dyadic Synthesis Filter Bank 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's outputs. These properties are described in "Output Characteristics (Setting the Output Parameter)" on page 7-194 of the Dyadic Analysis Filter Bank reference page.



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.

Examples

See the following DSP Blockset demos, which use the Two-Channel Synthesis Subband Filter block:

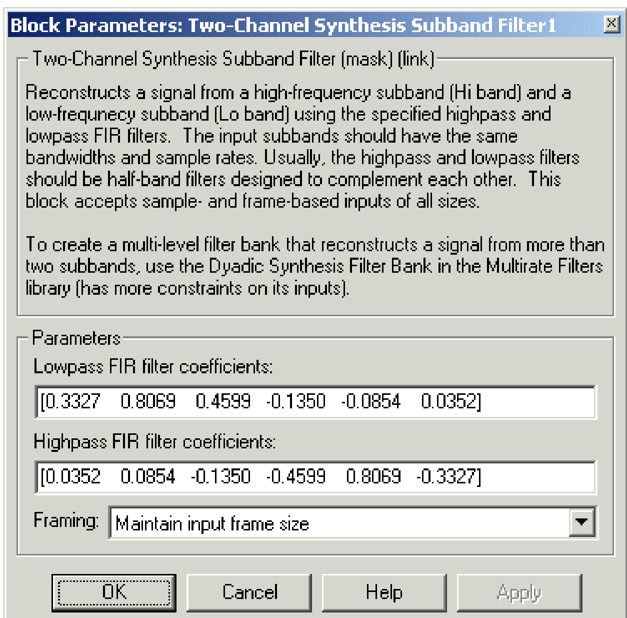
- Multi-level PR filter bank
- Denoising
- Wavelet transmultiplexer (WTM)

Note 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.

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



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, the filters in this block *must* be designed to perfectly reconstruct the outputs of the analysis filters. For more information, see "Specifying the FIR Filters" on page 7-602.

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, the filters in this block *must* be designed to perfectly reconstruct the outputs of the analysis filters. For more information, see "Specifying the FIR Filters" on page 7-602.

Framing

For frame-based inputs, the method by which to implement the interpolation: by doubling the output frame rate (**Maintain input frame size**), or doubling the output frame size (**Maintain input frame rate**). For more information, see "Frame-Based Operation" on page 7-594. Some settings of this parameter causes the block to have nonzero latency, as described in "Latency" on page 7-595.

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 to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

Dyadic Synthesis Filter Bank	DSP Blockset
Two-Channel Analysis Subband Filter	DSP Blockset
`fir1`	Signal Processing Toolbox
`fir2`	Signal Processing Toolbox
`firls`	Signal Processing Toolbox
`remez`	Signal Processing Toolbox

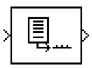
For related information, see “Multirate Filters” on page 4-32. Also see “Filtering” on page 7-6 for a list of all DSP Blockset filtering blocks.

Unbuffer

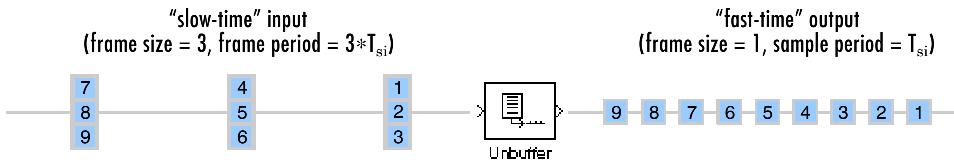
Purpose Unbuffer a frame input to a sequence of scalar outputs.

Library Signal Management / Buffers

Description

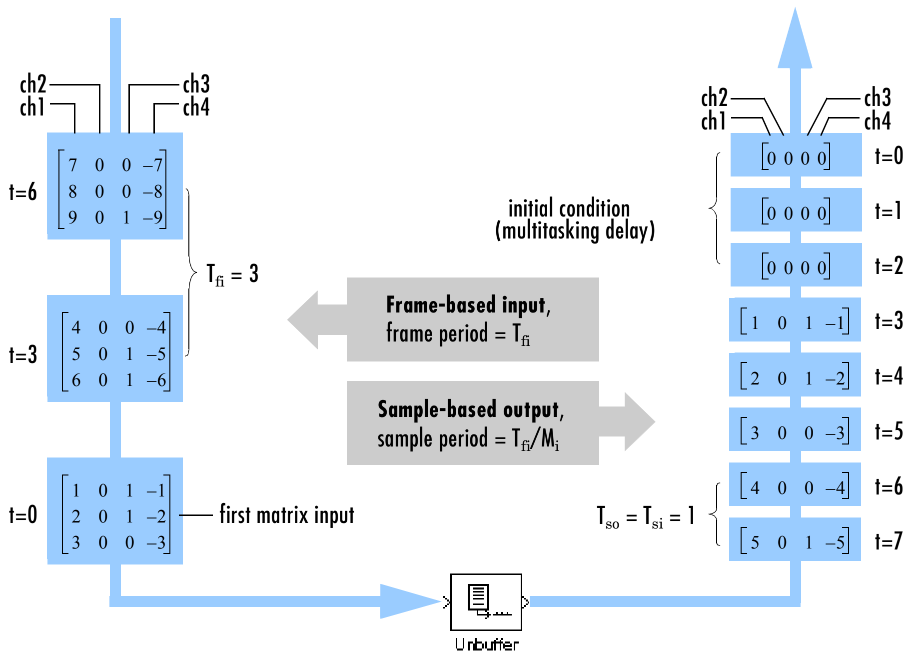


The Unbuffer block unbuffers an M_i-by-N frame-based input into a 1-by-N sample-based output. That is, inputs are unbuffered *row-wise* so that each matrix row becomes an independent time-sample in the output. The rate at which the block receives inputs is generally less than the rate at which the block 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 (see "Latency" below).



Latency

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.

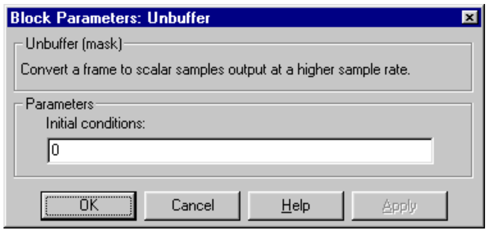
Nonzero Latency. For *multitasking* operation, the Unbuffer block's buffer is initialized with the value specified by the **Initial condition** parameter, and the block begins unbuffering this frame at the start of the simulation. Inputs to the block are therefore delayed by one buffer length, or M_i 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

See “Excess Algorithmic Delay (Tasking Latency)” on page 3-91 and “The Simulation Parameters Dialog Box” in the Simulink documentation for more information about block rates and the Simulink tasking modes.

Dialog Box



Initial conditions
The value of the block’s initial output for cases of nonzero latency; a scalar, vector, or matrix.

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 to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Buffer	DSP Blockset

See “Unbuffering a Frame-Based Signal into a Sample-Based Signal” on page 3-60 for related information. Also see “Buffers” on page 7-14 for a list of all the blocks in the Buffers library.

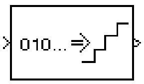
Purpose

Decode an integer input to a 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 B is specified by 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. If **Saturate** is selected, *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.

If **Wrap** is selected, *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
```

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.

Example

Consider a Uniform Decoder block with the following parameter settings:

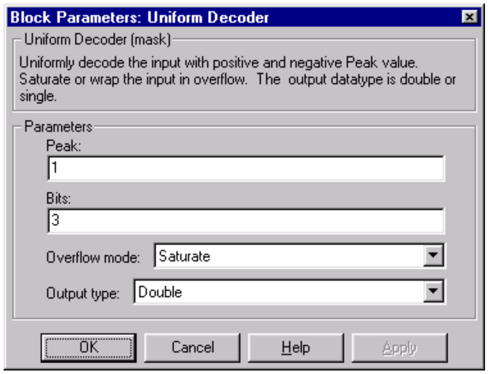
- **Peak** = 2
- **Bits** = 3

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 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

Supported Data Types

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

To learn how to convert to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

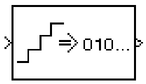
Data Type Conversion	Simulink
Quantizer	Simulink
Uniform Encoder	DSP Blockset
`udecode`	Signal Processing Toolbox
`uencode`	Signal Processing Toolbox

Also see "Quantizers" on page 7-13 for a list of all the blocks in the Quantizers library.

Purpose Quantize and encode a floating-point input to an 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 B is specified by the **Bits** parameter and V is specified by 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. If 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.

If 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 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 DSP 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.

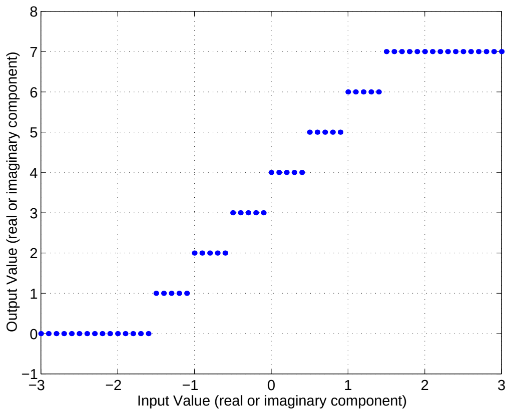
Bits	Unsigned Integer	Signed Integer
2 to 8	uint8	int8
9 to 16	uint16	int16
17 to 32	uint32	int32

The Uniform Encoder block operations adhere to the definition for uniform encoding specified in *ITU-T Recommendation G.701*.

Example

The figure below 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]`.

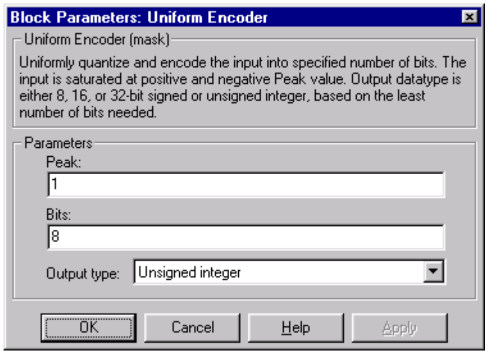
`-2.0`	is mapped to	`0`
`-1.5`	is mapped to	`1`
`-1.0`	is mapped to	`2`
`-0.5`	is mapped to	`3`
`0.0`	is mapped to	`4`
`0.5`	is mapped to	`5`
`1.0`	is mapped to	`6`
`1.5`	is mapped to	`7`

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`	
`-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 imag)

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 to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

Data Type Conversion	Simulink
Quantizer	Simulink
Uniform Decoder	DSP Blockset
`udecode`	Signal Processing Toolbox
`uencode`	Signal Processing Toolbox

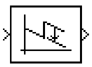
Also see “Quantizers” on page 7-13 for a list of all the blocks in the Quantizers library.

Unwrap

Purpose Unwrap the phase of a signal.

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 7-622
- "The Two Unwrap Modes" on page 7-623
- "Unwrap Method" on page 7-626
- "Definition of Phase Unwrap" on page 7-626

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. • Sample- or frame-based • M-by-N 2-D matrix or a 1-D vector	• Output elements are phase values in radians. • Same frame status as input • Same size and dimension as input • Output port rate = input port rate

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.

<table>
<tr><th colspan="2">Two Unwrap Modes</th></tr>
<tr><td colspan="2">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 7-626.) The number of times that k is reset to 0 depends on the unwrap mode.</td></tr>
<tr><th>Default Unwrap Mode:
Initialize k to 0 For Only the First Input Frame</th><th>Nondefault Unwrap Mode:
Set k to 0 For Each Successive Input Matrix or Input Vector</th></tr>
<tr><td>☐ 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.</td><td>☑ Do not unwrap phase discontinuities between successive frames

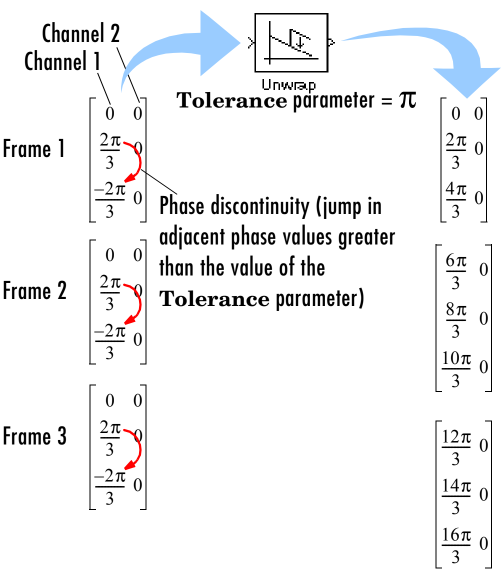
In this mode, k is reset to 0 for each successive input matrix or input vector. As k gets updated, the value of k is only retained within the current input matrix or vector. That is, the block unwraps each input's channel by considering phase discontinuities in the current input matrix or input vector only, ignoring discontinuities in previous inputs.</td></tr>
<tr><td>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</td><td>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</td></tr>
<tr><td>See the following diagrams.</td><td>See the following diagrams.</td></tr>
</table>

The following diagrams illustrate how the two unwrap modes operate on various inputs.

Default Unwrap Mode Operation: ☐ Do not unwrap phase discontinuities between successive frames

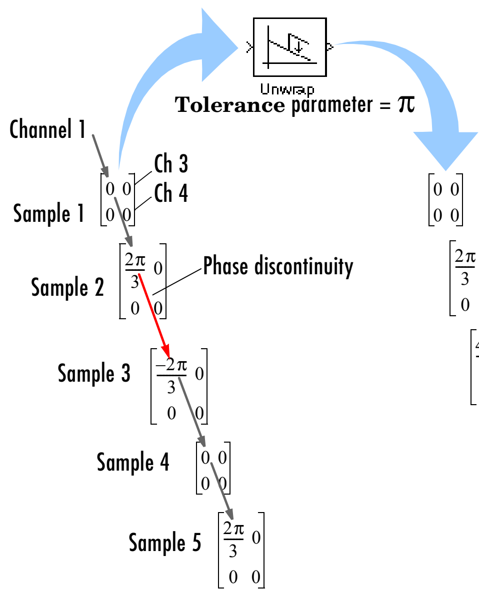
Frame-Based Inputs

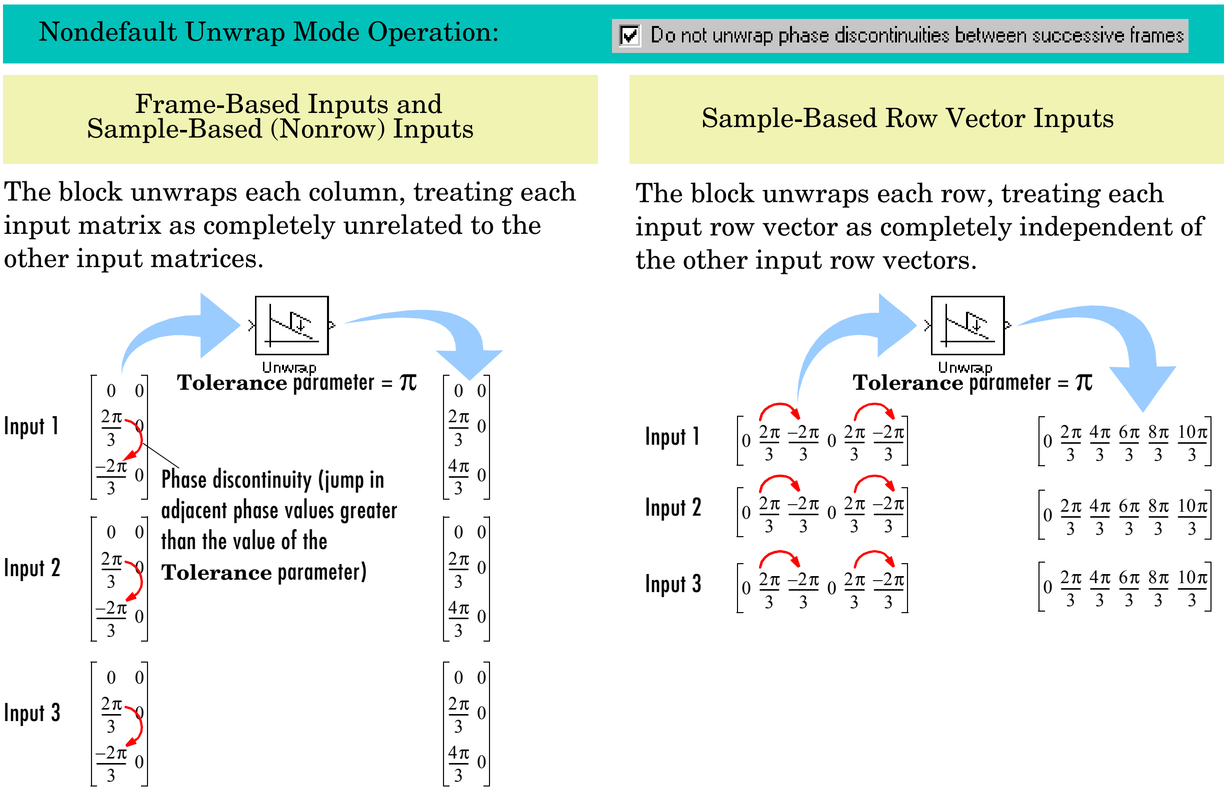
The block treats each input column as an independent channel. It unwraps by treating Channel 1 of Frame 2 as a continuation of Channel 1 of Frame 1.



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
Sample-Based Row Vector Inputs
The block unwraps each column, treating each input matrix as completely unrelated to the other input matrices.
The block unwraps each row, treating each input row vector as completely independent of the other input row vectors.
Unwrap
Tolerance parameter = π
Input 1
Input 2
Input 3
Phase discontinuity (jump in adjacent phase values greater than the value of the Tolerance parameter)
Unwrap
Tolerance parameter = π
Input 1
Input 2
Input 3

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.

Relevant Unwrap Terms:

- u_i — ith element of the input channel on which the algorithm operates
- α — **Tolerance** parameter value
- *phase jump* or *phase discontinuity* — difference between phase values of two adjacent channel entries that exceeds $|\alpha|$. The diagram in the next section indicates phase jumps with red arrows.

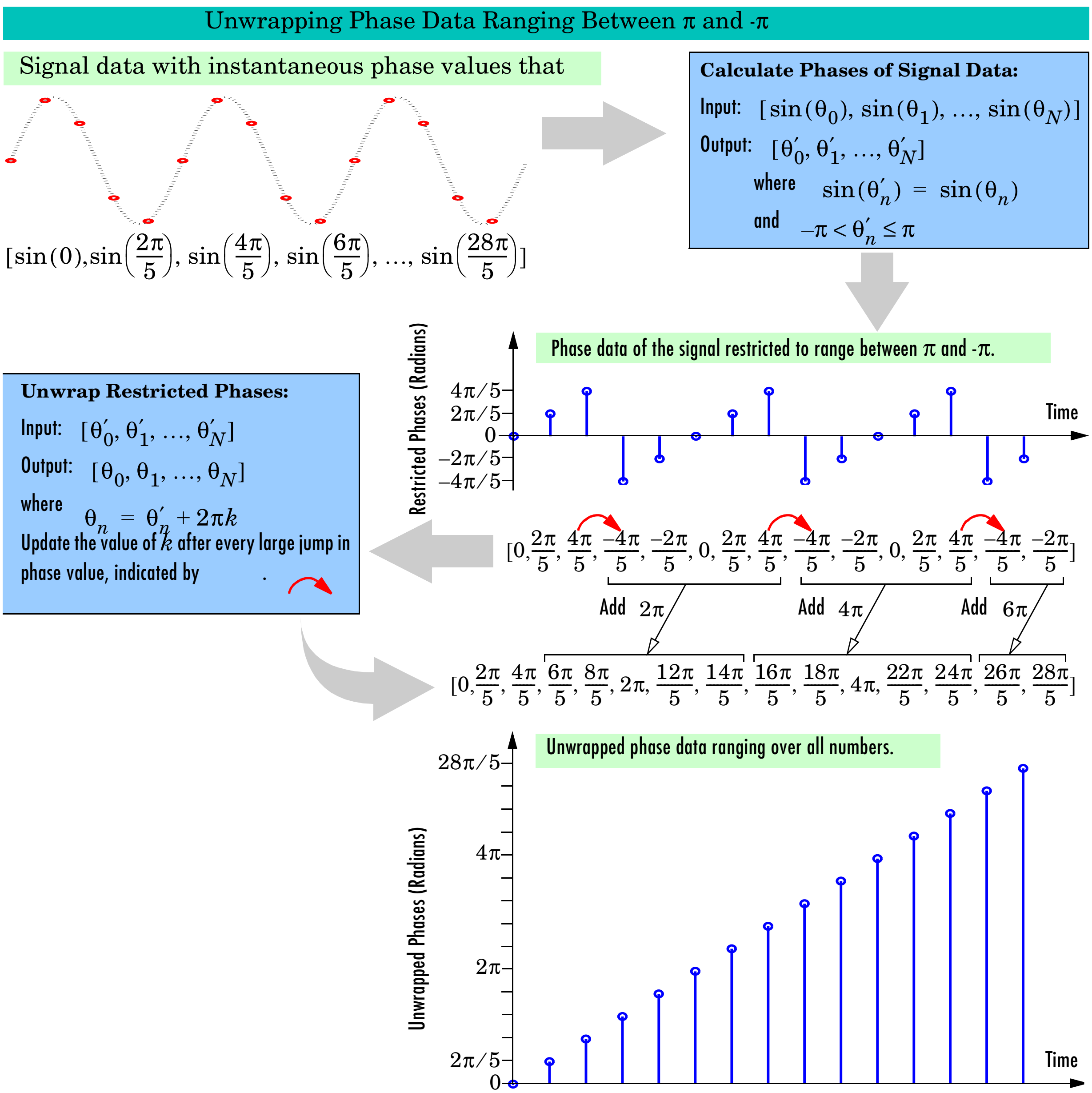
Steps to the Unwrap Method:

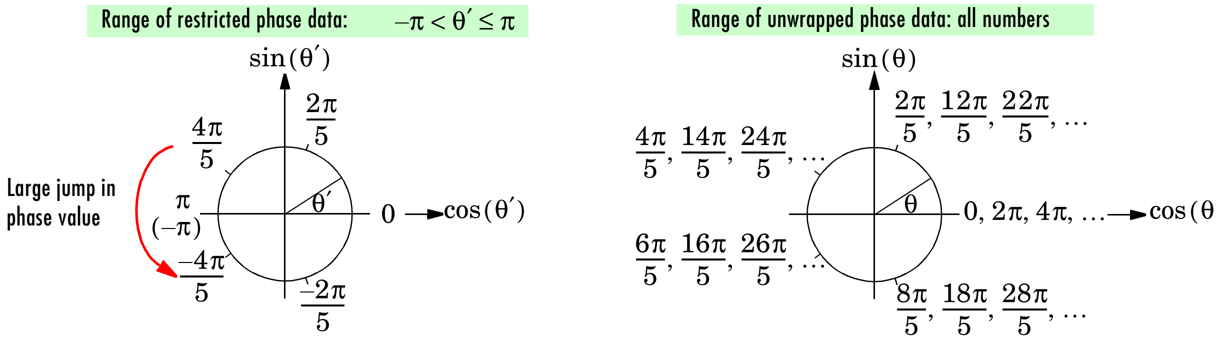
1. Set k to 0 (See "The Two Unwrap Modes" on page 7-623 for more on how often this step occurs.)
2. Check for a phase jump between adjacent channel elements u_i and u_{i+1}:
   - If there is no phase jump between u_i and u_{i+1} $(|u_{i+1} - u_i| \le |\alpha|)$, add $2\pi k$ to u_i, and then repeat step 2 to continue checking for phase jumps.
   - If there is a phase jump between u_i and u_{i+1} $(|u_{i+1} - u_i| > |\alpha|)$, add $2\pi k$ to u_i, and then go to step 3 to update k.
3. Update k as follows when there is a phase jump between u_i and u_{i+1}. Then go back to step 2 to add the updated $2\pi k$ value to u_{i+1} and succeeding channel elements until the next phase jump:
   - If $u_{i+1} < u_i$ (phase jump is negative), increment k.
   - If $u_{i+1} > u_i$ (phase jump is positive), decrement k.

Definition of Phase Unwrap

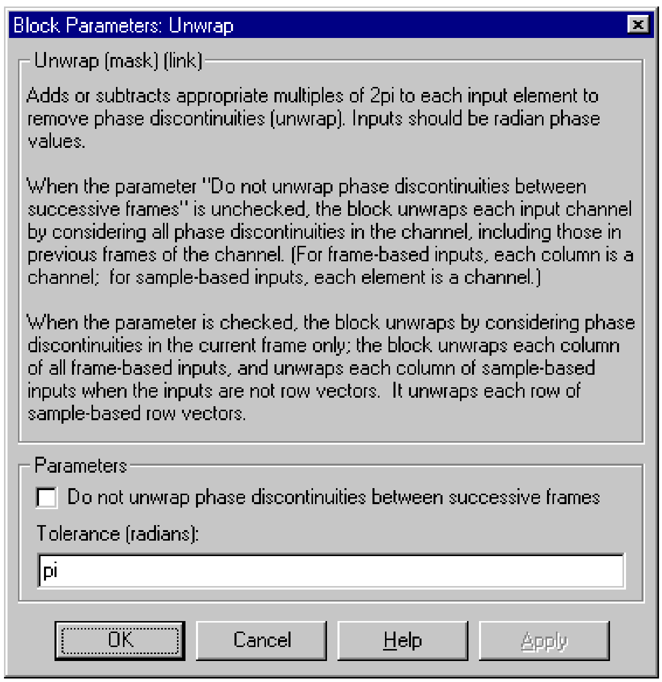
Algorithms that compute the phase of a signal often only output phases between $-\pi$ and π. For instance, such algorithms compute the phase of $\sin(2\pi + 3)$ to be 3, since $\sin(3) = \sin(2\pi + 3)$, and since the actual phase, $2\pi + 3$, is not between $-\pi$ and π. Such algorithms compute the 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 7-626.


Unwrapping Phase Data Ranging Between π and -π
Signal data with instantaneous phase values that
[sin(0),sin(2π/5), sin(4π/5), sin(6π/5), ..., sin(28π/5)]
Calculate Phases of Signal Data:
Input: [sin(θ0), sin(θ1), ..., sin(θN)]
Output: [θ'0, θ'1, ..., θ'N]
where sin(θ'n) = sin(θn)
and –π < θ'n ≤ π
Phase data of the signal restricted to range between π and -π.
Restricted Phases (Radians)
Time
Unwrap Restricted Phases:
Input: [θ'0, θ'1, ..., θ'N]
Output: [θ0, θ1, ..., θN]
where θn = θ'n + 2πk
Update the value of k after every large jump in phase value, indicated by .
Add 2π
Add 4π
Add 6π
Unwrapped phase data ranging over all numbers.
Unwrapped Phases (Radians)
Time



Dialog Box



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 this parameter is set, 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 7-623.

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 to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

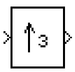
`unwrap`	MATLAB

Also see "Signal Operations" on page 7-16 for a list of all the blocks in the Signal Operations library.

Purpose Resample an input at a 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. The integer L is specified by the **Upsample factor** parameter. The **Sample offset** parameter delays the output samples by an integer number of sample periods D, where $0 \le D < (L-1)$, so that any of the L possible output phases can be selected.

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.

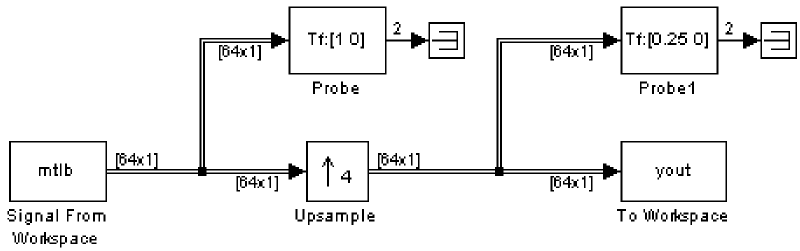
Frame-Based Operation

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.

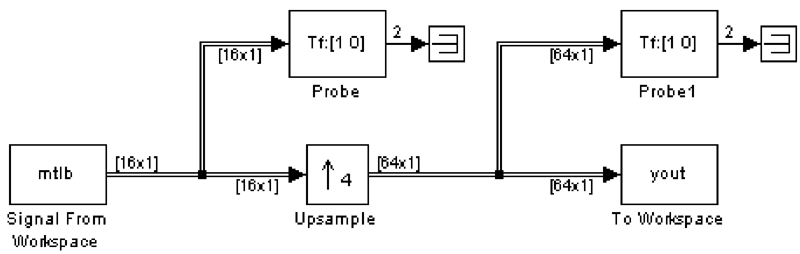
  The model below shows a single-channel input with a frame period of 1 second being upsampled by a factor of 4 to a frame period of 0.25 second. The input and output frame sizes are identical.



- **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.



Latency and Initial Conditions

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.

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.

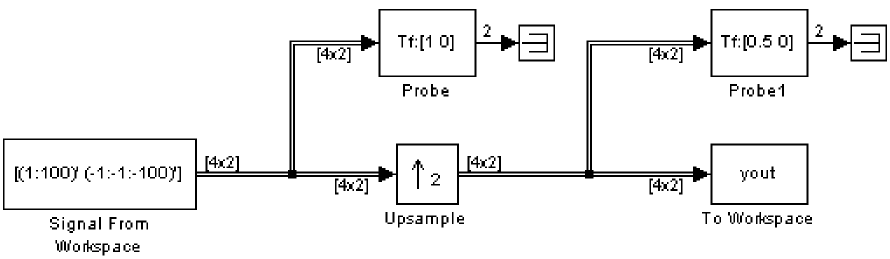
Zero tasking latency means that the block propagates the first input (received at t=0) immediately following the D consecutive zeros specified by the **Sample offset** parameter. This output (D+1) is followed in turn by the L-1 inserted zeros and the next input sample. The **Initial condition** parameter 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.

See "Excess Algorithmic Delay (Tasking Latency)" on page 3-91 and "The Simulation Parameters Dialog Box" in the Simulink documentation for more information about block rates and the Simulink tasking modes.

Example 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).

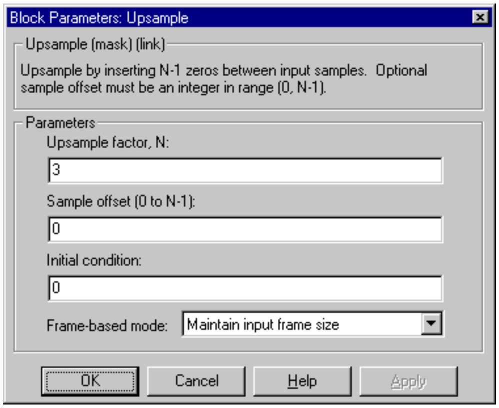
This model is multirate because there are at least two distinct frame rates, as shown by the two Probe blocks. To run this model in the Simulink multitasking mode, select **Fixed-step** and **discrete** from the **Type** controls in the **Solver** panel of the **Simulation Parameters** dialog box, and select **MultiTasking** from the **Mode** parameter. 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 (i.e., 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 (i.e., 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.

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** (i.e., increase the frame rate), or **Maintain input frame rate** (i.e., increase the frame size). The **Framing** parameter must be set to **Maintain input frame size** for sample-base inputs.

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 to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Downsample	DSP Blockset
FIR Interpolation	DSP Blockset
FIR Rate Conversion	DSP Blockset
Repeat	DSP Blockset

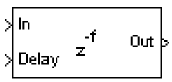
Also see “Signal Operations” on page 7-16 for a list of all the blocks in the Signal Operations library.

Variable Fractional Delay

Purpose Delay an input by a 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, is an M_i-by-N matrix of floating-point values in the range $0 \le v \le D$ that specifies the number of sample intervals to delay each channel of the input.

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 for the Variable Integer Delay block. See the section on sample-based initial conditions there for complete 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 $0 \leq v \leq D$ specifying the number of sample intervals to delay the current input. The input to the `Delay` port can be:

- 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 for the Variable Integer Delay block. See the section on frame-based initial conditions there for complete 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 (i.e., a higher order filter) yields a better estimate of the sample at the specified delay. A value between 4 and 6 for this parameter (i.e. 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 Q is specified by 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**.

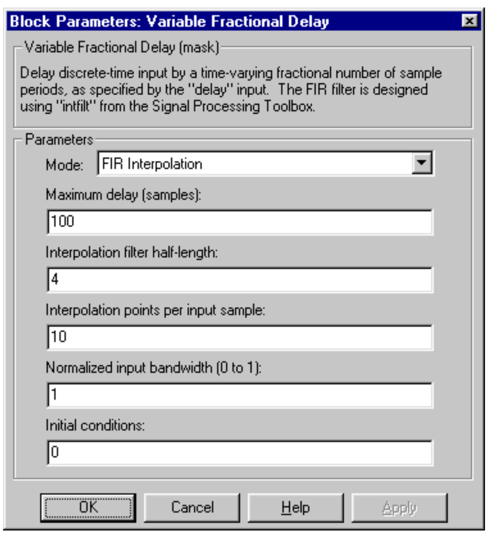
The block uses the `intfilt` function in the Signal Processing Toolbox to compute the FIR filters.

Note When the Variable Fractional Delay block is used in a feedback loop, at least one block with nonzero delay (e.g., an Integer Delay block with **Delay** > `0`) should be included in the loop as well. This prevents the occurrence of an algebraic loop if 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.

Dialog Box



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 to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Integer Delay	DSP Blockset
Unit Delay	Simulink
Variable Integer Delay	DSP Blockset

Also see “Signal Operations” on page 7-16 for a list of all the blocks in the Signal Operations library.

Variable Integer Delay

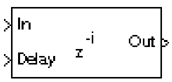
Purpose

Delay the input by a 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 `Delay` port input rate must be an integer multiple of the `In` port input rate. 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 Integer Delay block in the following ways.

Variable Integer Delay	Integer Delay
Delay is provided as an input to the `Delay` port.	Delay is specified as a parameter setting in the dialog box.
Delay can vary with time; for example, for a frame-based input, the *n*th element's delay in the first input frame can differ from the *n*th element's delay in the second input frame.	Delay cannot vary with time; for example, for a frame-based input, the *n*th 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.

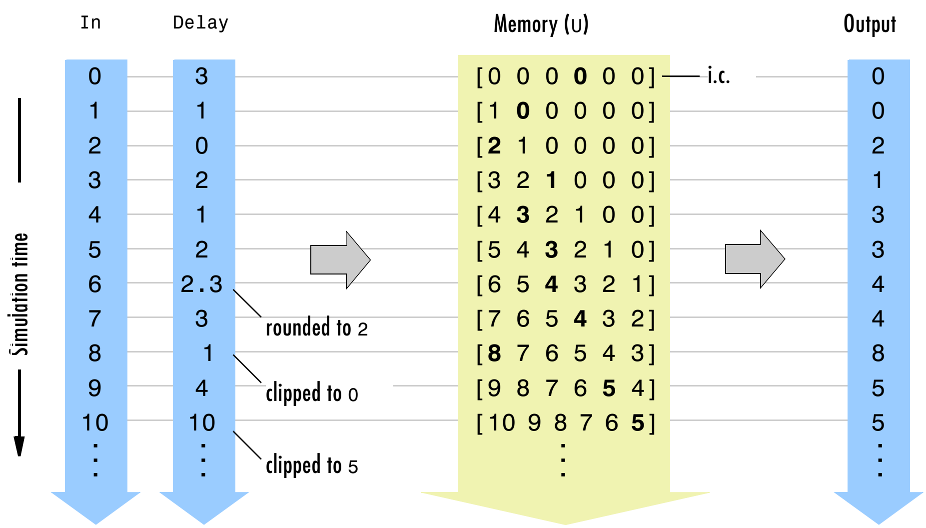
The Variable Integer Delay block stores the D+1 most recent samples received at the In port for each channel. At each sample time the block outputs the stored sample(s) indexed by the input to the Delay port.

For example, if 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. If 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.



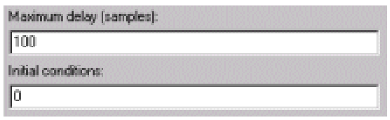
Note that the current input at each time-step is immediately stored in memory as U(1). This allows the current input to be available at the output for a delay of 0 (v=0).

The **Initial conditions** parameter specifies the values in the block's memory at the start of the simulation. Unlike the Integer 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,

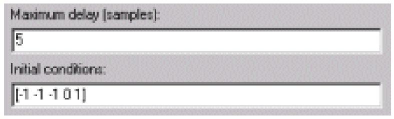
  the block initializes 100 M-by-N matrices in memory with zeros.

- *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" below for details.

Initial conditions cannot be specified by full matrices.

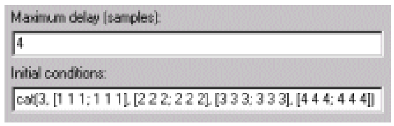
Time-Varying Initial Conditions. The following settings specify *time-varying* initial conditions. For a time-varying initial condition, the block initializes each of D samples in memory to one of the values entered in the **Initial conditions** parameter. This allows you to specify a unique output value 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]`.

- *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 matrix. For a 2-by-3 input and the parameters below,



the block initializes memory locations `U(2:5)` with values

$$U(2) = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}, U(3) = \begin{bmatrix} 2 & 2 & 2 \\ 2 & 2 & 2 \end{bmatrix}, U(4) = \begin{bmatrix} 3 & 3 & 3 \\ 3 & 3 & 3 \end{bmatrix}, U(5) = \begin{bmatrix} 4 & 4 & 4 \\ 4 & 4 & 4 \end{bmatrix}$$

An array initial condition can only be used with matrix inputs.

Initial conditions cannot be specified by full matrices.

Frame-Based Operation

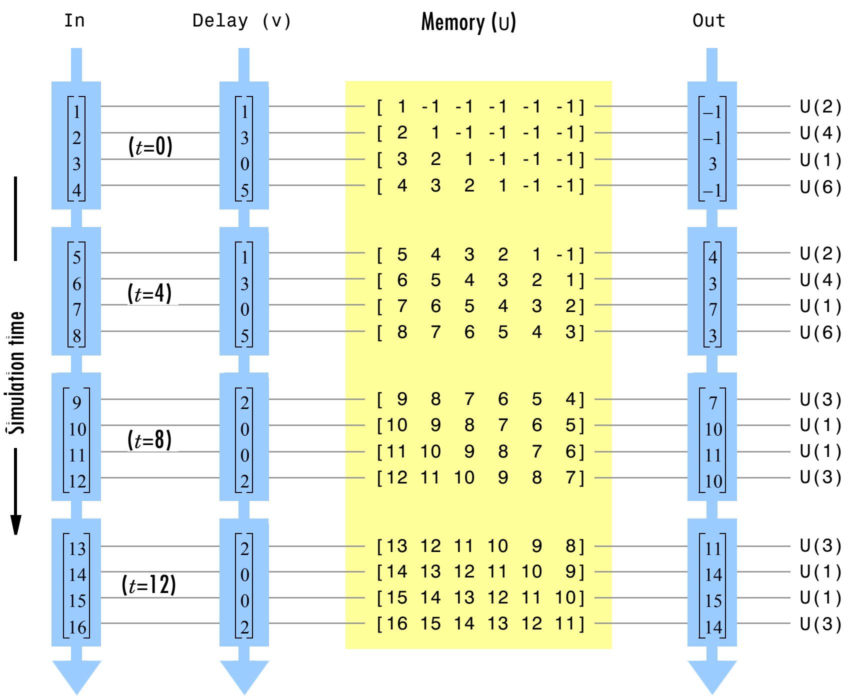
When the input is an M-by-N frame-based matrix, the block treats each of the N input columns as a frame containing M sequential time samples from an independent channel.

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-M 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-N 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.

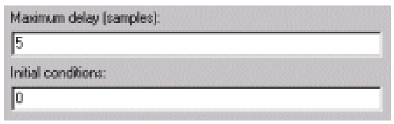
```
y(1) = U(v(1)+1)
y(2) = U(v(2)+1)
y(3) = U(v(3)+1)
y(4) = U(v(4)+1)
```



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.

- *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" 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 way:

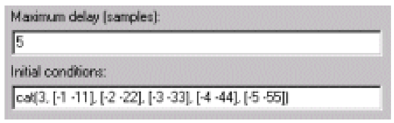
- *Vector* of dimensions 1-by-D. In this case, all channels have the same set of time-varying initial conditions specified by the entries of the vector. For the ramp input `[100; 100]'` with a frame size of 4, delay of 5, and the parameter settings below,

  the block outputs the following sequence of frames at the start of the simulation.

$$\begin{bmatrix} -1 & -1 \\ -2 & -2 \\ -3 & -3 \\ -4 & -4 \end{bmatrix}, \begin{bmatrix} -5 & -5 \\ 1 & 1 \\ 2 & 2 \\ 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 \\ 5 & 5 \\ 6 & 6 \\ 7 & 7 \end{bmatrix}, \ldots$$

- *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,



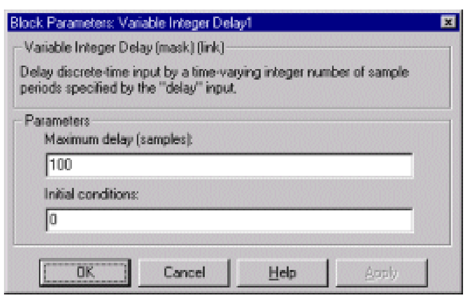
  the block outputs the following sequence of frames at the start of the simulation.

$$\begin{bmatrix} -1 & -11 \\ -2 & -22 \\ -3 & -33 \\ -4 & -44 \end{bmatrix}, \begin{bmatrix} -5 & -55 \\ 1 & 1 \\ 2 & 2 \\ 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 \\ 5 & 5 \\ 6 & 6 \\ 7 & 7 \end{bmatrix}, \ldots$$

  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.

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

- 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 to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

Integer Delay	DSP Blockset
Variable Fractional Delay	DSP Blockset

Also see "Signal Operations" on page 7-16 for a list of all the blocks in the Signal Operations library.

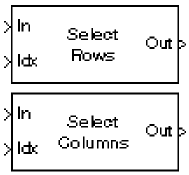
Purpose

Select a subset of rows or columns from the input.

Library

Signal Management / Indexing

Description



The Variable Selector block extracts a subset of rows or columns from the M-by-N input matrix at the `In` port, u.

When the **Select** parameter is set to **Rows**, the Variable Selector block extracts rows from the 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 u 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 u 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.

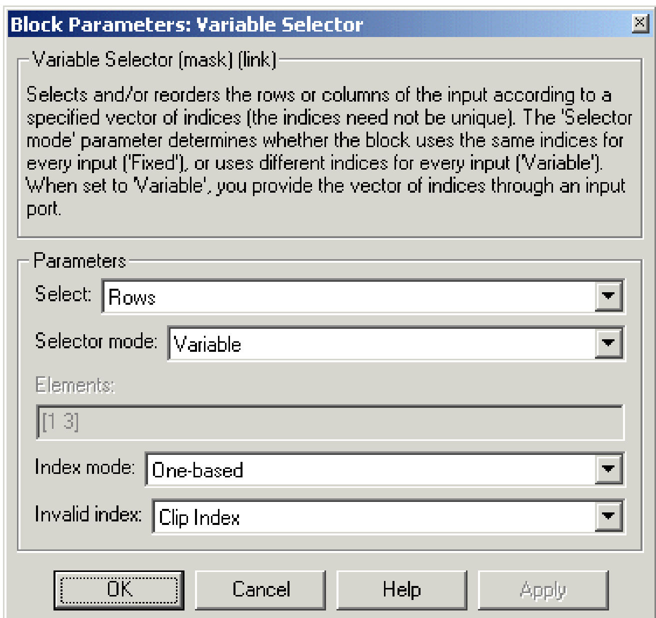
When the input is a 1-D vector, the **Select** parameter is ignored; the output is a 1-D vector of length L containing those elements specified by the length-L indexing vector.

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.

Note The Variable Selector block always copies the selected input rows to a contiguous block of memory (unlike the Simulink Selector block).

Dialog Box



Select

The dimension of the input to select, **Rows** or **Columns**. Tunable.

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 available when **Fixed** is selected in the **Selector mode** parameter. Tunable.

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.

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 to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

Multiport Selector	DSP Blockset
Permute Matrix	DSP Blockset
Selector	Simulink
Submatrix	DSP Blockset

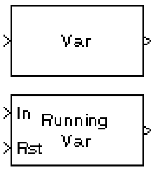
Also see "Indexing" on page 7-14 for a list of all the blocks in the Indexing library.

Variance

Purpose Compute the 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 the **Running variance** check box is *not* selected, 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.)

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:

$$y_j = \sigma_j^2 = \frac{\sum_{i=1}^{M} |u_{ij} - \mu_j|^2}{M-1} \qquad 1 \le j \le N$$

where μ_j is the mean of the *j*th column. 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 frame status of the output is the same as that of the input.

Running Operation

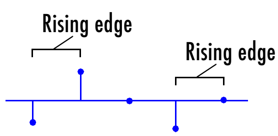
When the **Running variance** check box is selected, the block tracks the variance of each channel in a *time-sequence* of M-by-N inputs. For sample-based inputs, the output is a sample-based M-by-N matrix with each element y_{ij} containing the variance of element u_{ij} over all inputs since the last 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 jth column over all inputs since the last reset, up to and including element u_{ij} of the current input.

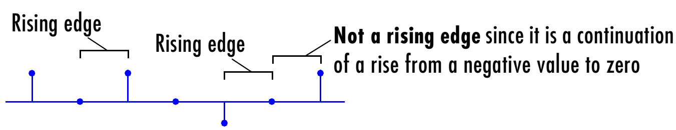
As in basic operation, length-M 1-D vector inputs and *sample-based* length-M row vector inputs are both treated as M-by-1 column vectors.

Resetting the Running 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.

The reset event is specified by the **Reset port** parameter, and can be one of the following:

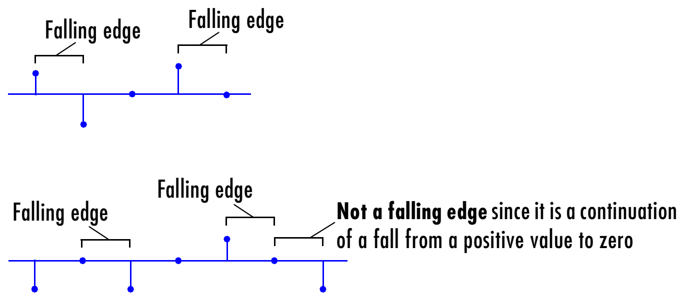
- **None** disables the `Rst` port.
- **Rising edge** — Triggers a reset operation when the `Rst` input does one of the following:
  - Rises from a negative value to a positive value or zero
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value 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)

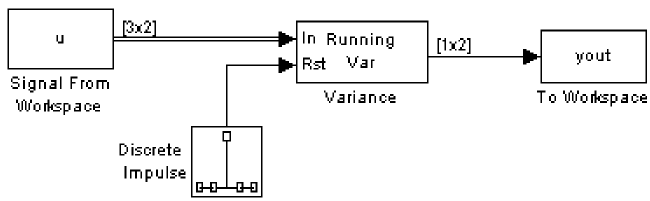


- **Either edge** — Triggers a reset operation when the `Rst` input is a **Rising edge** or **Falling edge** (as described above).
- **Non-zero sample** — Triggers a reset operation at each sample time that the `Rst` input is not zero.

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 3-91 and the topic on the **Simulation Parameters** dialog box in the Simulink documentation.

Example

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.



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`

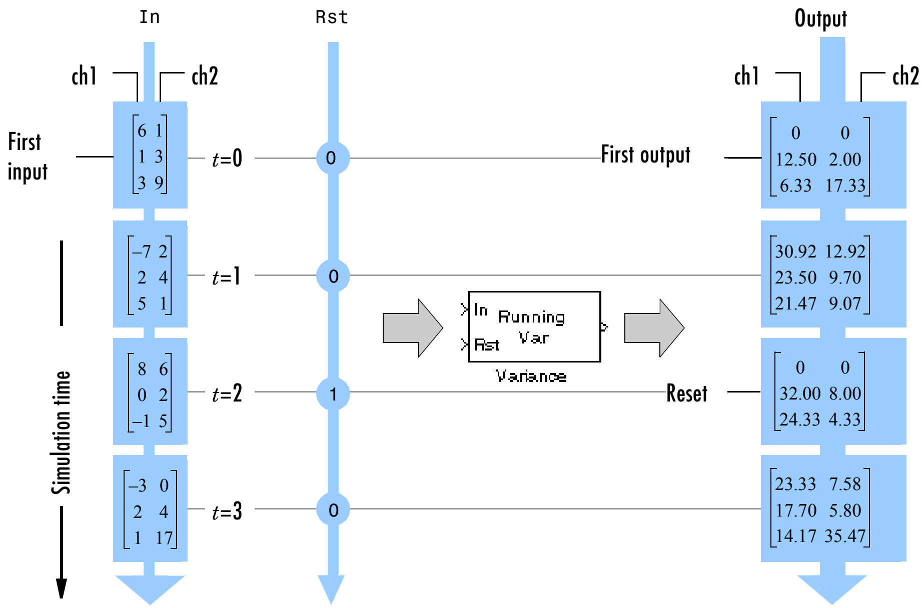
where

```
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:

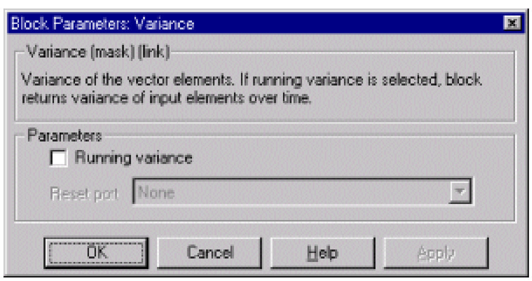
- **Delay (samples)** = `2`
- **Sample time** = `1`
- **Samples per frame** = `1`

The block's operation is shown in the figure below.



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

Dialog Box



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 set the **Running variance** parameter. For more information, see “Resetting the Running Variance” on page 7-655.

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 to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Mean	DSP Blockset
RMS	DSP Blockset
Standard Deviation	DSP Blockset
`var`	MATLAB

Also see “Statistics” on page 7-18 for a list of all the blocks in the Statistics library.

Vector Scope

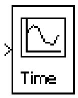
Purpose

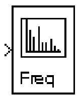
Display a vector or matrix of time-domain, frequency-domain, or user-defined data.

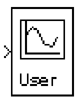
Library

DSP 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 short-time fast Fourier transform (STFFT) of a signal with a single block, use the Spectrum Scope block.

The input to the Vector Scope 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

Sections of This Reference Page

- "Specifying the Input Domain"
- "Changing the Display Span of the x-Axis"
- "Scaling the Horizontal Axis for Time-Domain Signals"
- "Scaling the Horizontal Axis for User-Defined Signals"
- "Scaling the Horizontal Axis for Frequency-Domain Signals"
- "Scope Properties"
- "Display Properties"
- "Axis Properties"
- "Line Properties"
- "Scope Window"
- "Dialog Box"
- "Supported Data Types"
- "See Also"

Specifying the Input Domain

Specify the domain of the input data as **Time**, **Frequency**, or **User-defined** in the **Input domain** parameter under the **Scope properties** check box.

Input Domain Parameter Settings

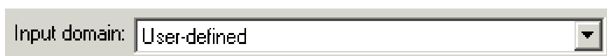
- **Time**

  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.

- **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, if 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)`.

- **User-defined**

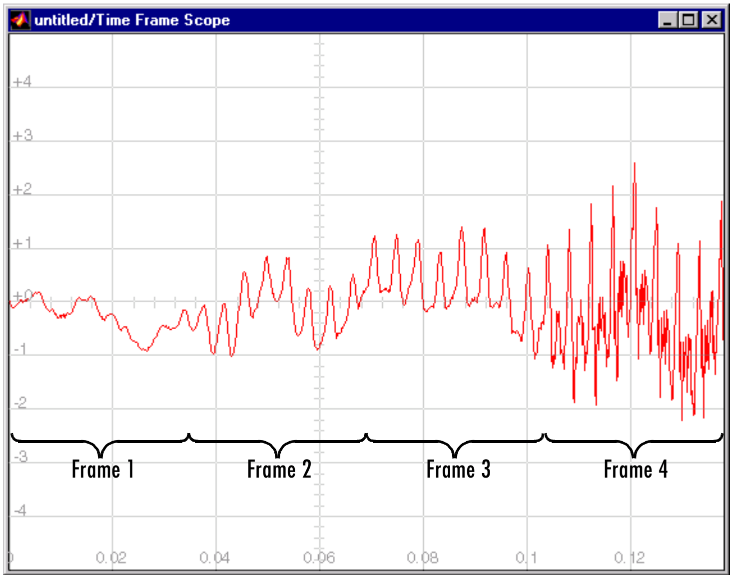


  For inputs specified as user-defined data, the block does not make any assumptions about the nature of the data in the input frame. In particular, it does not assume that it is time-domain or frequency-domain data. Various block parameters give you complete freedom to plot the data in the most appropriate manner.

Changing the Display Span of the x-Axis

The scope displays frames of data, and updates the display for each new input frame. The number of sequential frames displayed on the scope is specified by the **Time display span** parameter for time-domain signals, and the **Horizontal display span** parameter for user-defined signals. Setting either parameter to 1 plots the current input frame's data across the entire width of the scope. Setting these display-span parameters to larger numbers allows you to see a broader section of the signal by fitting more frames of data into the display region. A single frame is the smallest unit that can be displayed, so neither parameter can be less than 1.

Vector Scope Display with Time Display Span = 4



Scaling the Horizontal Axis for Time-Domain Signals

Scaling of the horizontal (time) axis for time-domain signals is automatic. 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** parameter. The spacing between time points is $T_{fi}/(M\text{-}1)$.

Scaling the Horizontal Axis for User-Defined Signals

To correctly scale the horizontal axis for user-defined signals, the block needs to know the spacing of the data in the input. This is specified by 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. For example, an input signal sampled at 500 Hz has an increment per sample of 0.002 second. The actual units of this interval (seconds, meters, Volts, etc.) are not needed for axis scaling.

When the **Inherit sample increment from input** check box is selected, 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, if 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 if 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 other cases, the frame rate and frame size do not provide enough information for the block to correctly scale the horizontal axis, and you should specify the appropriate value for the **Increment per sample in input frame** parameter. 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 parameter**.

Scaling the Horizontal Axis for Frequency-Domain Signals

In order 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. This is specified by the **Sample time of original time series** parameter.

When the **Inherit sample time from input** check box is selected, 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 **Frequency units** parameter specifies whether the frequency axis values should be in units of **Hertz** or **rad/sec**, and 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.

The Vector Scope block assumes that the input data spans the range $[0,F_s)$, as does 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.

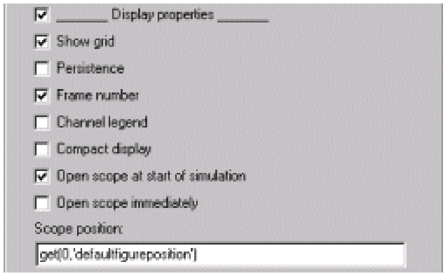
If the **Frequency units** parameter specifies **Hertz**, the spacing between frequency points is $1/(M*T_s)$. For **Frequency units** of **rad/sec**, the spacing between frequency points is $2\pi/(M*T_s)$. The **Amplitude scaling** parameter allows you to select **Magnitude** or **dB** scaling along the *y*-axis.

Scope Properties

The Vector Scope block allows you to plot time-domain, frequency-domain, or user-defined data, and adjust the frame span of the plot. Selecting the **Scope Properties** check box displays the **Input domain parameter**, which specifies the domain of the input data. In addition, for time-domain data, a **Time display span parameter** allows you to specify the number of frames to be displayed across the width of the scope window at any given time. For user-defined data, a **Horizontal display span parameter** serves the same function. Both of these parameters must be 1 or greater. See “Specifying the Input Domain” on page 7-661 for more information.

Display Properties

The Vector Scope and Spectrum Scope blocks offer a similar collection of display property settings. These can be exposed in the parameter dialog box by selecting the **Display properties** check box. Many of the properties can be accessed under the **Axes** menu in the *unzoomed* scope view (when **Compact display** is cleared), or by right-clicking on the scope window.



The **Show grid** parameter toggles the background grid on and off. This option can also be set in the **Axes** menu of the scope window.

When **Persistence** is selected, 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. This option can also be set in the **Axes** menu of the scope window.

When **Frame number** is selected, the number of the current frame in the input sequence is displayed on the scope window, incrementing the count as each new input is received. Counting starts at 1 with the first input frame, and continues until the simulation stops.

When **Channel legend** is selected, a legend indicating the line color, style, and marker of each channel's data is added. If the input signal is labeled, that label is displayed in the channel legend. If 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 on the legend to reposition it in the scope window; double click on the line label to edit the text. Note that when the simulation is rerun, the new edits are lost and the labels revert to the defaults. The **Channel legend** option can also be set in the **Axes** menu of the scope window.

When **Compact display** is selected, the scope completely fills the containing figure window. Menus and axis titles are not displayed, and the numerical axis labels are shown within the axes. When **Compact display** is cleared, the axis labels and titles are displayed in a gray border surrounding the scope axes, and the window's menus (including **Axes** and **Channels**) and toolbar are visible. This option can also be set in the **Axes** menu of the scope window.

When **Open scope at start of simulation** is selected, the scope opens at the start of the simulation. When this parameter is cleared, the scope does not open automatically during the simulation. To view the scope, double-click on the Vector Scope block, which brings up the scope as well as the block parameter dialog box. This feature is useful when you have several scope blocks in a model, and you do not want to view all the associated scopes during the simulation.

Open scope immediately allows you to open the scope from the Vector Scope parameters dialog box while the simulation is running. If the simulation is running and the scope window is not visible, you can double-click on the scope block to expose the scope window and the parameters dialog box. If you close the scope window during simulation, you can make it visible again by checking the **Open scope immediately** check box as long as the simulation is running. The check box will become cleared as soon as the scope opens.

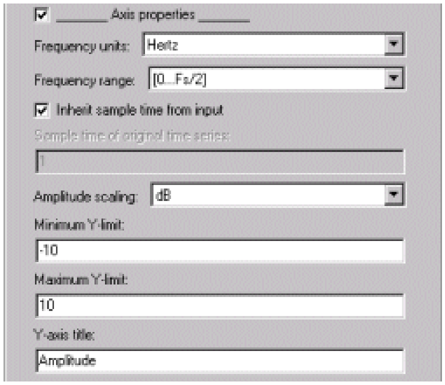
The **Scope position** parameter specifies a four-element vector of the form

```
[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` command for more information.

Axis Properties

The Vector Scope and Spectrum Scope blocks also share a similar collection of axis property settings. For the Vector Scope, the parameters listed under the **Axis properties** check box vary with the domain of the input. The dialogue box below shows the parameters available for frequency-domain data.



Minimum Y-limit and **Maximum Y-limit** set the range of the vertical axis. If **Autoscale** is selected from the right-click pop-up menu or from the **Axes** menu option, the **Minimum Y-limit** and **Maximum Y-limit** values are automatically recalculated to best fit the range of the data on the scope. Both of these parameters are available for all input domains.

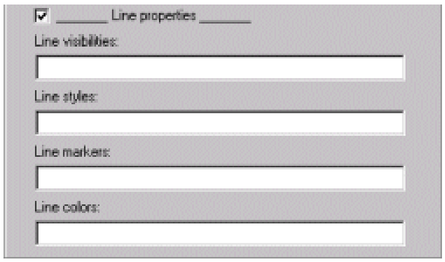
Y-axis title is the text to be displayed to the left of the y-axis. This parameter is available for all input domains. **X-axis title** is an analogous parameter available only when plotting user-defined data (this parameter is not visible in the dialog box shown).

Frequency-domain and user-defined data need extra information to scale the horizontal axis. For user-defined data, the parameters that provide this information are **Inherit sample increment from input and Increment in sample in input frame**. See "Scaling the Horizontal Axis for User-Defined Signals" on page 7-662 for more information. For frequency-domain data, an analogous pair of parameters, **Inherit sample time from input and Sample time of original time series**, must be specified. See "Scaling the Horizontal Axis for Frequency-Domain Signals" on page 7-663 for more information.

Three other parameters related to scaling the *x*-axis for frequency-domain signals are **Frequency units**, **Frequency range**, and **Amplitude scaling**. These are also described in "Scaling the Horizontal Axis for Frequency-Domain Signals" on page 7-663.

Line Properties

Both the Vector Scope and Spectrum scope also offer a similar collection of line property settings. These can be exposed in the parameter dialog box by selecting the **Line properties** check box. These properties can also be accessed under the **Channels** menu in the *unzoomed* scope view (when **Compact display** is cleared), or by right-clicking on the scope window.



The **Line properties** settings are typically used to help distinguish between two or more independent channels of data on the scope, as described in the following sections.

Line Visibilities. The **Line visibilities** parameter specifies which channels' 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, `|`.

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
```

channel 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).

Line Styles. 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

```
- | -- | : | -. | -
```

channel 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 `LineStyle` property of the `line` function in the MATLAB documentation for more information about the style syntax. To specify a marker for the individual sample points, use the **Line markers** parameter, described below.

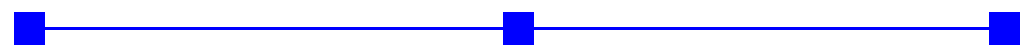
Line Markers. The **Line markers** parameter specifies the marker style with which each channel's samples are represented on the scope. The syntax specifies the channels' marker styles in list form, with each list entry specifying a marker for the corresponding channel's data. The list entries are separated by the pipe symbol, `|`.

For example, a five-channel signal would ordinarily generate all five plots with no marker symbol (i.e., 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
```

channel 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 first (leftmost) list item, `'*'`, corresponds to the first signal channel (leftmost column of the input matrix). See the `Marker` property of the `line` function in the MATLAB documentaion for more information about the available markers.

Type the word `stem` instead of one of the basic `Marker` shapes to produce a *stem plot* for the data in a particular channel.

Line Colors. 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]
```

channel 1 ch 2 ch 3 ch 4 ch 5

or

```
'k' | 'b' | 'r' | 'g' | [.7529 0 .7529]
```

channel 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)	
Green	(0,255,0)	
Dark purple	(192,0,192)	

Note that the first (leftmost) list item, `'k'`, corresponds to the first signal channel (leftmost column of the input matrix). See `ColorSpec` in the online MATLAB documentaion for more information about the color syntax.

Scope Window

The scope title (in the window title bar) is the same as the block title. The axis scaling is set by parameters listed under the **Axis properties** check box in the dialog box.

In addition to the standard MATLAB figure window menus (**File**, **Edit**, **Window**, **Help**), the Vector Scope window has an **Axes** and a **Channels** menu.

The properties listed in the **Axes** menu apply to all channels. Many of the parameters in this menu are also accessible through the block parameter dialog box. These are **Persistence**, **Show grid**, **Compact display**, **Frame number**, and **Channel legend**; see “Display Properties” on page 7-664 for more information. Below are descriptions of the other parameters listed 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 change them by editing those values.

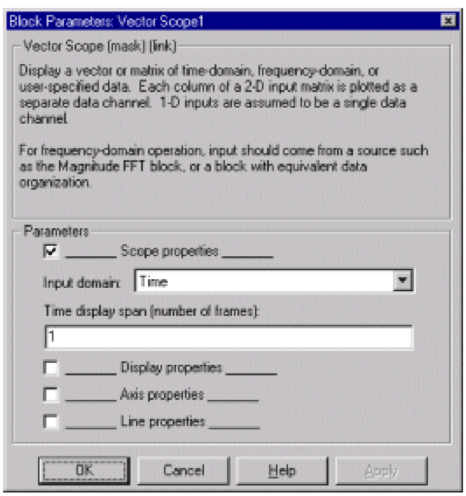
- **Save Position** automatically updates the **Scope position** parameter in the **Axis properties** field to reflect the scope window's current position and size. To make the scope window open at a particular location on the screen when the simulation runs, simply drag the window to the desired location, resize it as needed, 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. The parameters listed in this menu are **Visible**, **Style**, **Marker**, and **Color**; they correspond to the parameters listed in the dialog box under the **Line properties** check box. See "Line Properties" on page 7-668 for more information.

Many of these options can also be accessed by right-clicking with the mouse anywhere on the scope display. The menu that pops up contains a combination of the options available in both the **Axes** and **Channels** menus. The right-click menu is very helpful when the scope is in zoomed mode, when the **Axes** and **Channels** menus are not visible.

Dialog Box

Scope Properties Dialog Box



Scope properties

Select to expose **Scope properties** panel. See “Scope Properties” on page 7-664. Tunable.

Input domain

The domain of the input; **Time**, **Frequency**, or **User-defined**. See “Specifying the Input Domain” on page 7-661. Tunable.

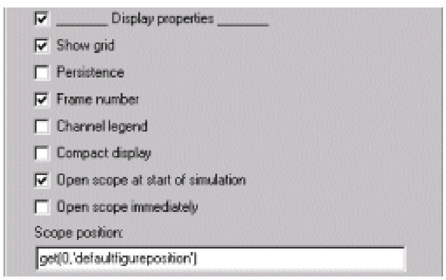
Time display span

The number of consecutive frames to display (horizontally) on the scope at any one time. (Visible when the **Input domain** parameter is **Time**.) See “Changing the Display Span of the x-Axis” on page 7-661.

Horizontal display span

(Not visible in the dialog box shown; appears under **Scope properties** when the **Input domain** parameter is **User-defined**.) The number of consecutive frames to display (horizontally) on the scope at any one time. See “Changing the Display Span of the x-Axis” on page 7-661.

Display Properties Dialog Box



Display properties

Select to expose **Display properties** panel. See “Display Properties” on page 7-664. Tunable.

Show grid

Toggles the scope grid on and off. See “Display Properties” on page 7-664. Tunable.

Persistence

Causes the window 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. See "Display Properties" on page 7-664. Tunable.

Frame number

Displays the number of the current frame in the input sequence, when selected with **Compact display** off. The frame number is not shown when **Compact display** is selected. See "Display Properties" on page 7-664. Tunable.

Channel legend

Toggles the legend on and off. See "Display Properties" on page 7-664. Tunable.

Compact display

Resizes the scope to fill the window. See "Display Properties" on page 7-664. Tunable.

Open scope at start of simulation

Opens the scope at the start of the simulation. When this parameter is cleared, the scope will not open automatically during the simulation; to view the scope, double click on the Vector Scope block during the simulation. This will bring up the scope as well as the block parameter dialog box. See "Display Properties" on page 7-664. Tunable.

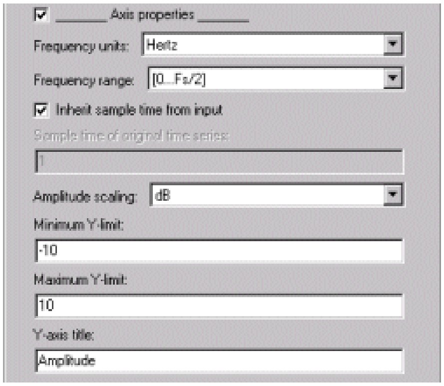
Open scope immediately

Opens the scope from the Vector Scope parameters dialog box while the simulation is running. The check box becomes cleared automatically after use. See "Display Properties" on page 7-664. Tunable.

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. See "Display Properties" on page 7-664. Tunable.

Axis properties Dialog Box



Axis properties

Select to expose the **Axis Properties** panel. See “Axis Properties” on page 7-666. Tunable.

Frequency units

The frequency units for the *x*-axis, **Hertz** or **rad/sec**. (Visible when the **Input domain** parameter is **Frequency**.) See “Axis Properties” on page 7-666. Tunable.

Frequency range

The frequency range over which to plot the data, **[0..Fs/2]**, **[-Fs/2..Fs/2]**, or **[0..Fs]**, where F_s is the sample frequency of the original time-domain signal, $1/T_s$. (Visible when the **Input domain** parameter is **Frequency**.) See “Axis Properties” on page 7-666. Tunable.

Inherit sample time from input

Computes the time-domain sample period from the frame period and frame size of the frequency-domain input; use only if 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 is was generated. (Visible when the **Input domain** parameter is **Frequency**.) See “Axis Properties” on page 7-666. Tunable.

Sample time of original time series

The sample period of the original time-domain signal, T_s. (Visible when the **Input domain** parameter is **Frequency**.) See “Axis Properties” on page 7-666. Tunable.

Inherit sample increment from input

(Not visible in the dialog box shown; appears under **Axis properties** when the **Input domain** parameter is **User-defined**.) Scales the horizontal axis by computing the horizontal interval between samples in the input frame from the frame period of the input; use only if the input’s sample period is equal to the period with which the physical signal was originally sampled. See “Axis Properties” on page 7-666. Tunable.

Increment per sample in input frame

(Not visible in the dialog box shown; appears under **Axis properties** when the **Input domain** parameter is **User-defined**.) The numerical interval between adjacent x-axis points corresponding to the user-defined input data. See “Axis Properties” on page 7-666. Tunable.

Amplitude scaling

The scaling for the y-axis, **dB** or **Magnitude**. (Visible when the **Input domain** parameter is **Frequency**.) See “Axis Properties” on page 7-666. Tunable.

Minimum Y-limit

The minimum value of the y-axis. Tunable.

Maximum Y-limit

The maximum value of the y-axis. Tunable.

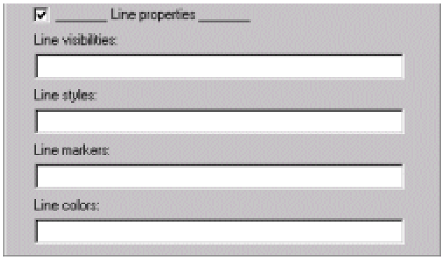
Y-Axis title

The text to be displayed to the left of the y-axis. Tunable.

X-Axis title

(Not visible in the dialog box shown; appears under **Axis properties** when the **Input domain** parameter is **User-defined**.) The text to be displayed below the x-axis. Tunable.

Line Properties Dialog Box



Line properties

Select to expose the **Line Properties** panel. See “Line Properties” on page 7-668. Tunable.

Line visibilities

The visibility of the various channels’ scope traces, `on` or `off`. Channels are separated by a pipe (|) symbol. See “Line Properties” on page 7-668. Tunable.

Line styles

The line styles of the various channels’ scope traces. Channels are separated by a pipe (|) symbol. See “Line Properties” on page 7-668. Tunable.

Line markers

The line markers of the various channels’ scope traces. Channels are separated by a pipe (|) symbol. See “Line Properties” on page 7-668. Tunable.

Line colors

The colors of the various channels’ scope traces, in one of the `ColorSpec` formats. Channels are separated by a pipe (|) symbol. See “Line Properties” on page 7-668. Tunable.

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 to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

Matrix Viewer	DSP Blockset
Spectrum Scope	DSP Blockset

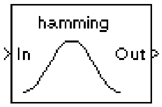
Also see the following topics:

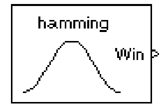
- "Viewing Signals" on page 3-80 — How to use this and other blocks to view signals
- "DSP Sinks" on page 7-3 — List of all blocks in the DSP Sinks library

Purpose Compute a window, and/or apply a window to an input signal.

Library Signal Operations

Description





The Window Function block has three modes of operation, selected by the **Operation** parameter as described below.

Operation Modes

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 the vector element-wise with each of the N channels in the M-by-N input matrix *u*.

  ```
  y = repmat(w,1,N) .* u          % Equivalent MATLAB code
  ```

  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. If 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.

- **Generate and apply window**

  In this mode the block computes an M-by-1 window vector, *w*, and multiplies the vector element-wise with each of the N channels in the M-by-N input matrix *u*.

  ```
  y = repmat(w,1,N) .* u          % Equivalent MATLAB code
  ```

  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. If the input is frame-based, output *y* is frame-based; otherwise, 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 Sampling

For the generalized-cosine windows (**Blackman**, **Hamming**, and **Hann**), the **Sampling** parameter determines whether the window samples are computed in a *periodic* or a *symmetric* manner. For example, if **Sampling** is set to **Symmetric**, a Hamming window of length M is computed as

```
w = hamming(M)     % Symmetric (aperiodic) window
```

If **Sampling** is set to **Periodic**, the same window is computed as

```
w = hamming(M+1)   % Periodic (asymmetric) window
w = w(1:M)
```

Window Type

The available window types are shown in the table below. The **Stopband attenuation in dB** and **Beta** parameters specify the characteristics of the **Chebyshev** and **Kaiser** windows, respectively, and are only available when those window designs are selected.

When **Window type** is set to **User defined**, the Window function block computes the user-defined window specified by the **Window function name** parameter. If the user-defined window requires parameters other than the window length, select the **Additional parameters for user defined window** check box. The cell array entered in **Window function parameters** determines the values of the additional parameters.

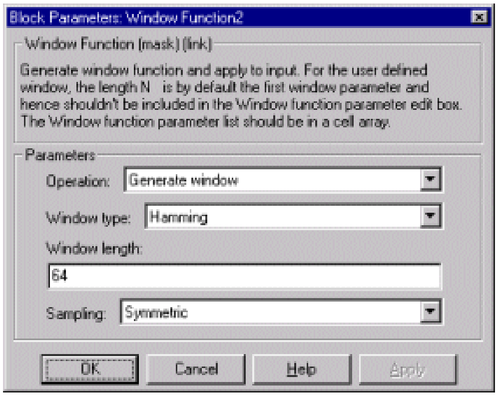
For complete information about the other window functions, consult the Signal Processing Toolbox documentation.

Window Type	Description
Bartlett	Computes a Bartlett window. `w = bartlett(M)`
Blackman	Computes a Blackman window. `w = blackman(M)`
Rectangular	Computes a rectangular window. `w = rectwin(M)`

Window Type	Description
Chebyshev	Computes a Chebyshev window with stopband ripple `R`. `w = chebwin(M,R)`
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`}`

Dialog Box



Operation

The block's operation: **Apply window to input**, **Generate window**, or **Generate and apply window**. The input/output port configuration is updated to match the parameter setting.

Window type

The type of window to apply. Tunable.

Window length

The length of the window to apply. This parameter is available only when **Generate window** is selected in the **Operation** menu. Otherwise, the window vector length is computed to match the input frame size, M.

Sampling

The window sampling for generalized-cosine windows, **Symmetric** or **Periodic**. Tunable.

Stopband attenuation in dB

(Not shown in dialog above. Visible for the **Chebyshev** window.) The level (dB) of stopband attenuation, R_s. Tunable.

Beta

(Not shown in dialog above. Visible for the **Kaiser** window.) 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.

Window function name

(Not shown in dialog above. Visible for **User defined** windows.) The name of the user-defined window function to be calculated by the block.

Additional parameters for user defined window

(Not shown in dialog above. Visible for **User defined** windows.) Enables the **Window function parameters** when selected. Select when the user-defined window requires parameters other than the window length.

Window function parameters

(Not shown in dialog above.Visible for **User defined** windows.) The extra parameters required by the user-defined window function (besides the window length), enabled when **Additional parameters for user defined window** is selected. The entry must be a cell array.

Supported Data Types

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

To learn how to convert to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

FFT	DSP 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
`kaiser`	Signal Processing Toolbox
`triang`	Signal Processing Toolbox

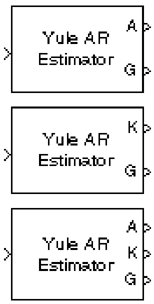
Also see “Signal Operations” on page 7-16 for a list of all the blocks in the Signal Operations library.

Yule-Walker AR Estimator

Purpose Compute an estimate of AR model parameters using the 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 **Inherit estimation order from input dimensions** is selected, 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. The Yule-Walker AR Estimator and Burg AR Estimator blocks return similar results for large frame sizes.

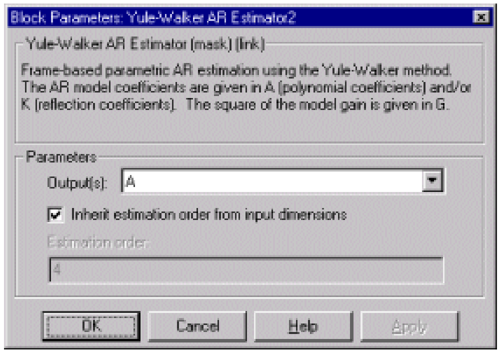
When **Output(s)** is set to **A**, port `A` is enabled. Port `A` outputs a column vector of length p+1 that contains the normalized estimate of the AR model coefficients in descending powers of z,

```
[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`.

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**). Tunable.

Inherit estimation order from input dimensions

When selected, sets the estimation order *p* to one less than the length of the input vector. Tunable.

Estimation order

The order of the AR model, *p*. This parameter is enabled when **Inherit estimation order from input dimensions** is not selected.

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

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

To learn how to convert to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Burg AR Estimator	DSP Blockset
Covariance AR Estimator	DSP Blockset
Modified Covariance AR Estimator	DSP Blockset
Yule-Walker Method	DSP Blockset
`aryule`	Signal Processing Toolbox

Also see “Parametric Estimation” on page 7-5 for a list of all the blocks in the Parametric Estimation library.

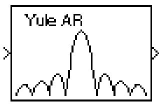
Purpose

Compute a parametric estimate of the spectrum using the Yule-Walker 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.

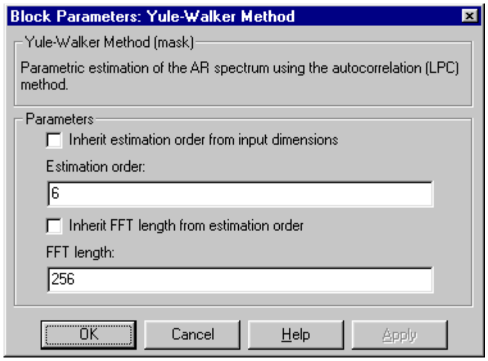
The input is a sample-based vector (row, column, or 1-D) or frame-based vector (column only) representing a frame of consecutive time samples from a single-channel signal. The block's output (a column vector) is the estimate of the signal's power spectral density at N_{fft} equally spaced frequency points in the range $[0,F_s)$, where F_s is the signal's sample frequency.

When **Inherit estimation order from input dimensions** is selected, 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 **Inherit FFT length from estimation order** is selected, N_{fft} is specified by (estimation order + 1), which must be a power of 2. When **Inherit FFT length from estimation order** is *not* selected, 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 **Inherit estimation order from input dimensions** is not selected.

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. If N_{fft} exceeds the input frame size, the frame is zero-padded as needed. This parameter is enabled when **Inherit FFT length from estimation order** is not selected.

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

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

To learn how to convert to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-3.

See Also

Burg Method	DSP Blockset
Covariance Method	DSP Blockset
Levinson-Durbin	DSP Blockset
Autocorrelation LPC	DSP Blockset
Short-Time FFT	DSP Blockset
Yule-Walker AR Estimator	DSP Blockset
`pyulear`	Signal Processing Toolbox

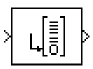
See “Power Spectrum Estimation” on page 6-6 for related information. Also see a list of all blocks in the Power Spectrum Estimation library.

Zero Pad

Purpose Alter the input size by zero-padding or truncating rows and/or columns.

Library Signal Operations

Description



The Zero Pad block changes the size of the input matrix from M_i-by-N_i to M_o-by-N_o by zero-padding or truncating along the rows, the columns, or both dimensions. The dimensions of the output, M_o and N_o, are specified by the **Number of output rows** and **Number of output columns** parameters, respectively. You can set **Action when truncation occurs** so that the block gives a warning or an error when truncation occurs.

The **Zero pad along** parameter specifies how the input should be altered. The options are:

- **Columns**

  When **Columns** is selected, the **Number of output rows** parameter (M_o) is enabled, and the block pads or truncates each input *column* by an equal amount. If M_o>M_i, the block pads by adding M_o-M_i rows of zeros to the bottom of the matrix. If M_o<M_i, the block truncates by deleting M_i-M_o rows from the bottom of the matrix. In both cases, the number of columns is unchanged (N_o=N_i). A 1-D vector input is zero padded or truncated at the "bottom," and the output is a 1-D vector.
- **Rows**

  When **Rows** is selected, the **Number of output columns** parameter (N_o) is enabled, and the block pads or truncates each input *row* by an equal amount. If N_o>N_i, the block pads by adding N_o-N_i columns of zeros to the right side of the matrix. If N_o<N_i, the block truncates by deleting N_i-N_o columns from the right side of the matrix. In both cases, the number of rows is unchanged (M_o=M_i). A 1-D vector input is zero padded or truncated at the "bottom," and the output is a 1-D vector.
- **Columns and rows**

  When **Columns and rows** is selected, both the **Number of output rows** parameter (M_o) and the **Number of output columns** parameter (N_o) are enabled, and the block pads or truncates rows and columns as specified. A length-M_i 1-D vector input is treated as an M_i-by-1 matrix and the output is an M_o-by-N_o matrix.

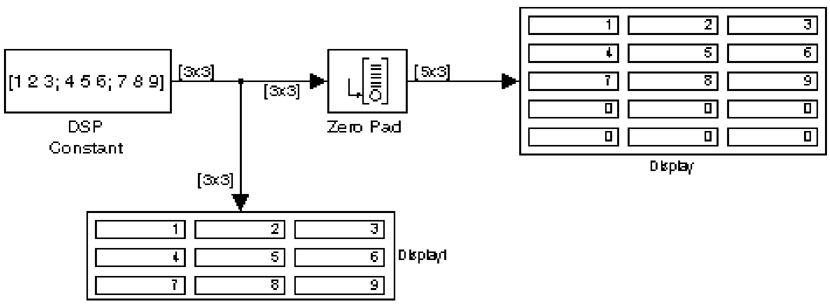
- **None**

  When **None** is selected, the input is passed through to the output without padding or truncation.

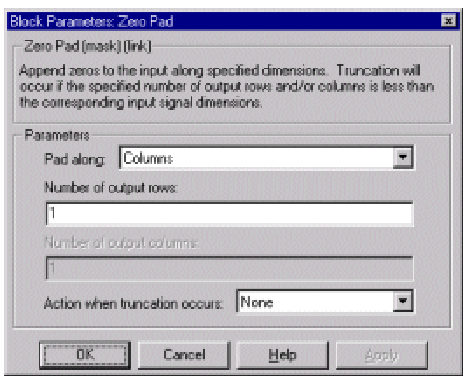
Example

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:

- **Zero pad along** = **Columns**
- **Number of output rows** = 5



Dialog Box



Zero 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 desired number of rows in the output, M_o. This parameter is enabled when **Columns** or **Columns and rows** is selected in the **Zero pad along** menu.

Number of output columns

The desired number of columns in the output, N_o. This parameter is enabled when **Rows** or **Columns and rows** is selected in the **Zero pad along** menu.

Action When Truncation Occurs

The block's behavior when the input matrix is truncated. It gives a **Warning**, an **Error**, or gives no indication of the truncation when set to **None**.

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 to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-3.

See Also

Matrix Concatenation	Simulink
Pad	DSP Blockset
Repeat	DSP Blockset
Submatrix	DSP Blockset
Upsample	DSP Blockset
Variable Selector	DSP Blockset

Also see "Signal Operations" on page 7-16 for a list of all the blocks in the Signal Operations library.

8

Function Reference

Functions — Alphabetical List

`dsp_links`	Display and return library link information
`dspfwiz`	Open the Filter Realization Wizard GUI
`dsplib`	Open DSP Blockset block library
`dspstartup`	Set default Simulink model parameters for DSP systems
`liblinks`	Display library link information for blocks linked to the DSP Blockset.
`rebuffer_delay`	Compute delay introduced by the Buffer and Unbuffer blocks

Purpose Display library link information for blocks linked to the DSP Blockset

Syntax

```
dsp_links
dsp_links(sys)
dsp_links(sys,mode)
ret_links = dsp_links(...)
```

Description `dsp_links` displays library link information for blocks linked to the DSP Blockset. For each block in the current model, `dsp_links` replaces the block name with the full pathname to the block's library link in the DSP Blockset. Blocks linked to v4 or later DSP Blockset blocks are highlighted in green while blocks linked to v3 DSP Blockset blocks are highlighted in yellow. Blocks at all levels of the model are analyzed.

A summary report indicating the number of blocks linked to each blockset version is also displayed in the MATLAB command window. The highlighting and link display is disabled when the model is executed or saved, or when `dsp_links` is executed a second time from the MATLAB command line.

`dsp_links(sys)` toggles the display of block links in system `sys`. If `sys` is the current model (`gcs`), this is the same as the plain `dsp_links` syntax.

`dsp_links(sys,`*`mode`*`)` directly sets the link display state, where *`mode`* can be `'on'`, `'off'`, or `'toggle'`. The default is `'toggle'`.

`ret_blks = dsp_links(...)` returns a structure, each field of which is a cell array that lists the full paths to blocks' library links in the DSP Blockset. The different fields refer to different versions of the libraries.

See Also

`liblinks`	DSP Blockset

dspfwiz

Purpose

Open the Filter Realization Wizard GUI

Syntax

```
dspfwiz
```

Description

`dspfwiz` opens the Filter Realization Wizard GUI, which is also accessible as a block in the Filter Designs library.

For more information on using the GUI, see the Filter Realization Wizard reference page.

See Also

Digital Filter	DSP Blockset
Digital Filter Design	DSP Blockset
Filter Realization Wizard	DSP Blockset

Purpose

Open the main DSP Blockset library

Syntax

```
dsplib
dsplib ver
```

Description

`dsplib` opens the current version of the main DSP Blockset library.

`dsplib` *`ver`* opens version *`ver`* of the DSP Blockset library, where *`ver`* can be `2`, `3`, or `4`.

When you launch an older version of the DSP Blockset, MATLAB displays a message reminding you that a newer version exists.

dspstartup

Purpose Configure the Simulink environment for DSP 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, "Using dspstartup.m" on page C-3.

The `dspstartup.m` script sets the following Simulink environment parameters. See Appendix A, "Model and Block Parameters," in the Simulink documentation for complete information about a particular setting.

Parameter	Setting
`SingleTaskRateTransMsg`	`error`
`Solver`	`fixedstepdiscrete`
`SolverMode`	`SingleTasking`

Parameter	Setting
`StartTime`	`0.0`
`StopTime`	`inf`
`FixedStep`	`auto`
`SaveTime`	`off`
`SaveOutput`	`off`
`AlgebraicLoopMsg`	`error`
`InvariantConstants`	`on`
`RTWOptions`	`[get_param(0,'RTWOptions')', -aRollThreshold=2']`

See Also

`startup` MATLAB

liblinks

Purpose Display library link information for blocks linked to the DSP Blockset

Syntax

```
liblinks
liblinks(sys)
liblinks(sys,mode,lib)
liblinks(sys,mode,lib,clrs)
blks = liblinks(...)
```

Description Please see the command line help for `liblinks`. Type

```
help liblinks
```

in the MATLAB command window.

See Also

`dsp_links`	DSP 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 **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` = **Buffer size** `m` = **Buffer overlap**

Unbuffer	`f` = input frame size
	`n = 1`
	`m = 0`

See Also

Buffer	DSP Blockset
Unbuffer	DSP Blockset

A

Data Type Support

Note All data type support applies to both simulation and Real-Time Workshop C code generation.

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

See the following sections for more information on data type support in the DSP Blockset:

- “Supported Data Types and How to Convert to Them” on page A-3 — Table of all data types supported by the DSP Blockset, including blocks and functions for converting to these data types
- “Viewing Data Types of Signals In Models” on page A-5 — How to enable automatic data type labeling of signals in models
- “Correctly Defining Custom Data Types” on page A-6 — How to define custom data types
- “Fixed-Point Support” on page A-7 — Description of the fixed-point data types and blocks that support them.
- “Boolean Support” on page A-10 — Benefits of using the Boolean data type, list of blocks supporting the Boolean data type, how to disable Boolean support, and consequences of disabling Boolean support.

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 DSP Blockset blocks support single- and double-precision floating point.

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

Supported Data Types and How to Convert to Them

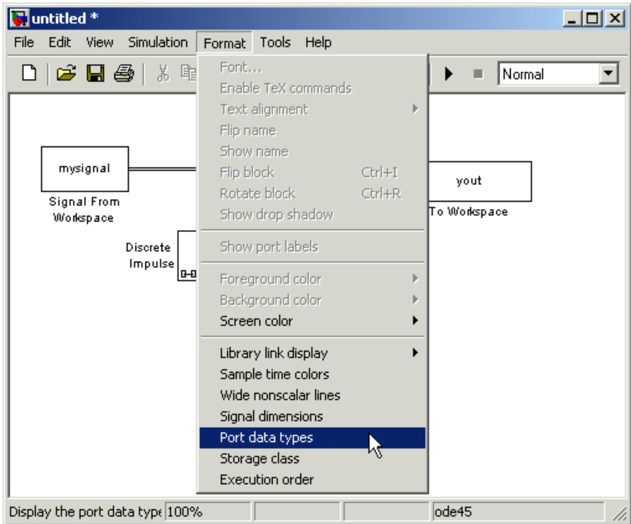
Data Types Supported by DSP Blockset Blocks	Commands and Blocks for Converting Data Types	Comments
Double-precision floating point	• `double` • Data Type Conversion block	Simulink built-in data type supported by all DSP Blockset blocks.
Single-precision floating point	• `single` • Data Type Conversion block	Simulink built-in data type supported by all DSP Blockset blocks.
Boolean	• `boolean` • Data Type Conversion block	Simulink built-in data type. To learn more, see "Boolean Support" on page A-10.
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)

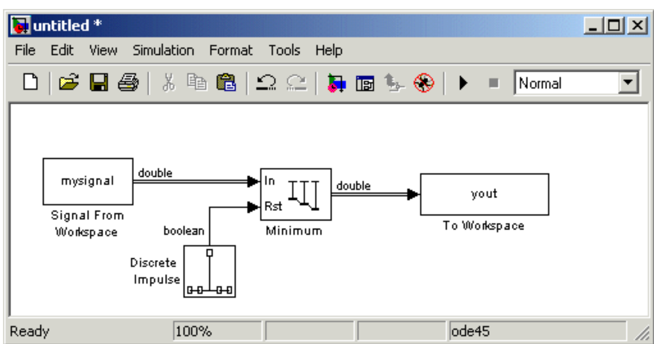
Data Types Supported by DSP Blockset Blocks	Commands and Blocks for Converting Data Types	Comments
Fixed-point data types (`sfix`, `ufix`, `sint`, `uint`, `sfrac`, `ufrac`)	• Blocks in the Data Type library of the Fixed-Point Blockset • Fixed-Point Blockset functions for conversions (listed in the online categorical function reference of Fixed-Point Blockset) • Functions and GUIs for designing quantized filters with the Filter Design Toolbox (compatible with Filter Realization Wizard block)	To learn more about fixed-point support, see "Fixed-Point Support" on page A-7.
Custom data types	See "Correctly Defining Custom Data Types" on page A-6 to learn about custom data types.	

Viewing Data Types of Signals In Models

To enable data type labels of the signals in a model, select the **Format** menu in the model and select **Port data types** as shown in Figure . The signal lines in the model will then have labels indicating their data types as shown in Figure . (To see the labels, you may have to refresh the model diagram by selecting the **Edit** menu in your model and then selecting **Update diagram**.



Enabling Data Type Labels of Signals



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 DSP Blockset blocks may not properly support your custom data types.

Fixed-Point Support

Many DSP Blockset blocks support fixed-point data types (including some source blocks). These blocks are compatible with blocks from the Fixed-Point Blockset, and with Simulink blocks that support fixed-point data types.

Note To take full advantage of the DSP Blockset fixed-point capabilities, you must install the Fixed-Point Blockset. For more information, see the topic on licensing information in the Fixed-Point Blockset documentation.

The following topics provide more information on fixed-point support:

- "Blocks Supporting Fixed-Point" on page A-7 — List of DSP Blockset blocks that support fixed-point data types
- "Implementing Fixed-Point Filters" on page A-9 — Implementing fixed-point filters with the Filter Realization Wizard block
- "Related Fixed-Point Topics" on page A-9 — Where to get more information about fixed-point data types

Blocks Supporting Fixed-Point

The following blocks are all of the blocks in the DSP Blockset that support fixed-point data types. These blocks are compatible with blocks from the Fixed-Point Blockset, and with Simulink blocks that support fixed-point data types. These blocks are colored orange in the DSP Blockset library. (Some of the blocks are Simulink blocks that are available in the DSP Blockset libraries as well as the Simulink libraries.)

To take full advantage of the fixed-point capabilities of the following blocks that are *not* Simulink blocks, you must install the Fixed-Point Blockset. For more information, see the topic on licensing information in the Fixed-Point Blockset documentation.

- Buffer
- Check Signal Attributes
- Constant Diagonal Matrix
- Convert 1-D to 2-D
- Convert 2-D to 1-D
- Create Diagonal Matrix
- Data Type Conversion (Simulink block)
- Delay Line
- Discrete Impulse
- Display (Simulink block)
- Downsample
- DSP Constant
- Edge Detector
- Extract Diagonal
- Extract Triangular Matrix
- Filter Realization Wizard
- Flip
- Frame Status Conversion
- Identity Matrix
- Inherit Complexity
- Integer Delay
- Matrix Concatenation (Simulink block)
- Matrix Viewer
- Maximum
- Minimum
- Multiphase Clock
- Multiport Selector
- N-Sample Enable
- N-Sample Switch
- Overwrite values
- Pad
- Permute Matrix
- Queue
- Repeat
- Sample and Hold
- Selector (Simulink block)
- Signal To Workspace
- Sine Wave
- Spectrum Scope
- Stack
- Submatrix
- Time Scope (Simulink block)
- Toeplitz
- Transpose
- Triggered Delay Line
- Triggered To Workspace
- Unbuffer
- Upsample
- Variable Integer Delay
- Variable Selector
- Vector Scope
- Zero Pad

Implementing Fixed-Point Filters

You can implement fixed-point filters with the Filter Realization Wizard block when you install the Filter Design Toolbox and the Fixed-Point Blockset. For more information, see the online reference page of the Filter Realization Wizard block.

Related Fixed-Point Topics

To learn more about fixed-point data types, see the following related documentation. (To access most of the features described in these documents, you must install the Fixed-Point Blockset, as described in the topic on licensing information in the Fixed-Point Blockset documentation.)

- Topic on fixed-point numbers in the Fixed-Point Blockset documentation — How the Fixed-Point Blockset and DSP Blockset represent fixed-point numbers (`sfix`, `ufix`, `sint`, `uint`, `sfrac`, `ufrac`)
- The following online reference sections in the Fixed-Point Blockset:
  - Blocks in the Data Type library — Description of blocks for converting data types in a model
  - References for `sfix`, `ufix`, `sint`, `uint`, `sfrac`, `ufrac` — Description of functions for creating MATLAB structures describing various fixed-point data types
  - Functions for conversions — Description of functions for converting floating-point to fixed-point numbers, converting legacy models to use fixed-point data types, etc.
- Topics in the Filter Design Toolbox documentation on working with quantized filters and quantizing filters in the Filter Design and Analysis Tool GUI — How to use Filter Design Toolbox to construct quantized filters that you can implement with the Filter Realization Wizard block
- Filter Realization Wizard block online block reference page — Detailed information about the Filter Realization Wizard block, including how to use it to implement fixed-point filters.

Boolean Support

Many DSP 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 a later section.)

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

- "Advantages of Using the Boolean Data Type"
- "Lists of Blocks Supporting Boolean Inputs or Outputs" on page A-10
- "Effects of Enabling and Disabling Boolean Support" on page A-11
- "Steps to Disabling Boolean Support" on page A-12

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 Appendix B, "Code Generation Support.")

Lists of Blocks Supporting Boolean Inputs or Outputs

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

- Counter
- Cumulative Product
- Cumulative Sum
- Histogram
- Maximum
- Mean
- Minimum
- N-Sample Enable
- N-Sample Switch
- RMS
- Standard Deviation
- Variance

The following blocks have input ports such as `Push`, `Pop`, and `Clear` that accept the Boolean data type:

- Queue
- Stack

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

- Counter
- Edge Detector
- Event-Count Comparator
- LPC to LSF/LSP Conversion
- LU Factorization
- Multiphase Clock
- N-Sample Enable
- Polynomial Stability Test
- Queue
- Stack

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 DSP 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

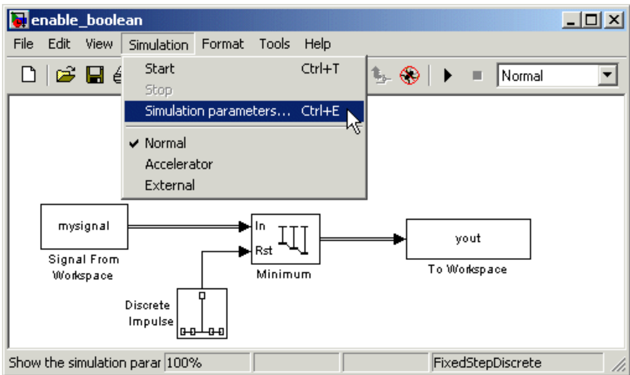
To disable Boolean data type support in a particular model, clear the Boolean-enabling simulation parameter in the model by completing the following:

- "Step 1: Open the Simulation Parameters Dialog Box" on page A-13
- "Step 2: Disable the Boolean Data Type in the Advanced Tab" on page A-14
- "Step 3: (Optional) Verify Data Types of Signals" on page A-14

You can also set Simulink simulation preferences so that *all* new models you create have Boolean support disabled. For more information, see the topic on setting advanced Simulink preferences in the Simulink documentation.

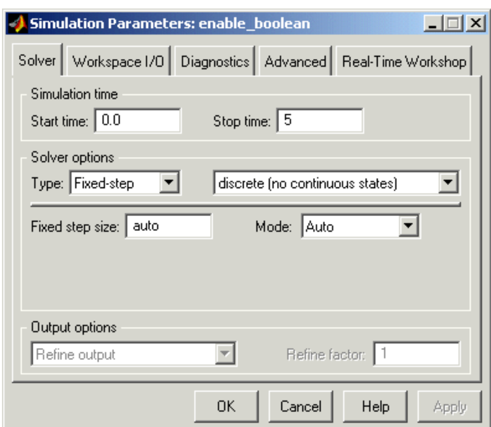
Step 1: Open the Simulation Parameters Dialog Box

In the model for which you want to enable Boolean data type support, open the **Simulation Parameters** dialog box by selecting the **Simulation** menu in your model and then selecting **Simulation parameters...**. as shown in Figure .



Opening the Simulation Parameters Dialog Box

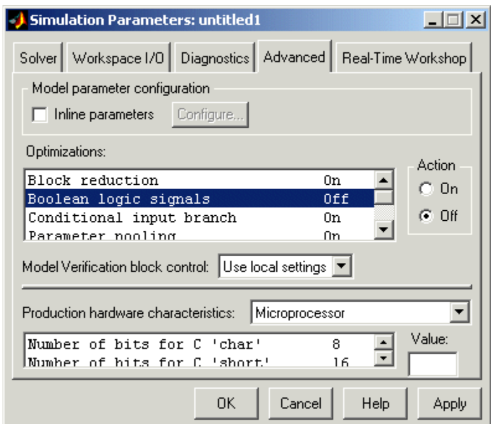
The following figure illustrates the **Simulation Parameters** dialog box with the appropriate settings for DSP simulations (note the discrete **Fixed-step** solver setting).



Step 2: Disable the Boolean Data Type in the Advanced Tab

Click the **Advanced** tab of the **Simulation Parameters** dialog, set the **Optimizations** parameter to **Boolean logic signals**, and set the **Action** check box to **Off**, as shown in Figure . 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 A-11.



Settings for Disabling the Boolean Data Type

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 A-5.) 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 A-11).

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**.

B

Code Generation Support

ANSI C Code Generation Support

You can generate ANSI C code from DSP Blockset blocks by using Real-Time Workshop (RTW) or Real-Time Workshop Embedded Coder. *All* DSP Blockset blocks support the following code generation targets:

- Generic real-time (GRT) — In Real-Time Workshop
- Embedded real-time (ERT) — In Real-Time Workshop Embedded Coder

The following topics provide more information on ANSI C code generation support:

- “Highly Optimized Generated C Code” on page B-2 — Descriptions of various optimizations made in C code generated from DSP Blockset blocks
- “Related C Code Generation Topics” on page B-3 — Where to get more information about generating C code from Simulink models

Highly Optimized Generated C Code

All DSP Blockset blocks generate highly optimized ANSI C code. This C code is often suitable for use in real-time embedded processors, and include the following optimizations:

- **Function reuse (run-time libraries)** — The generated code *reuses* common algorithmic functions via calls to *run-time functions*. Run-time functions are highly optimized ANSI C functions that implement core algorithms such as FFT and convolution. Run-time functions are precompiled into ANSI C *run-time libraries*, and enable the blocks to generate smaller, faster code that requires less RAM.
- **Parameter reuse (RTW run-time parameters)** — In many cases, if there are multiple instances of a block that all have the same value for a specific parameter, each block instance points to the same variable in the generated code. This reduces memory requirements.
- **Blocks have parameters for code optimization** — Various blocks provide parameters for specifying whether to optimize the generated code for memory or for speed (these optimizations also affect simulation). For example, the FFT and Sine Wave blocks provide this capability.

- **Other optimizations** — Use of contiguous input and output arrays, reusable inputs, overwritable arrays, and in-place algorithms provide smaller generated C code that is more efficient at run-time.

Related C Code Generation Topics

To learn more about ANSI C code generation, see the following related documentation:

- Real-Time Workshop documentation — How to use Real-Time Workshop to generate code from Simulink models
- Real-Time Workshop Embedded Coder documentation — How to use Real-Time Workshop Embedded Coder to generate code from Simulink models
- The topic on run-time parameters in the Simulink Writing S-Function documentation — How run-time parameters aid in generation of better C code

C

Configuring Simulink for DSP Systems

When you create a new DSP model, you may want to adjust certain Simulink settings to suit your own needs. A typical change, for example, is to adjust the **Stop time** parameter (in the **Simulation Parameters** dialog box) to a different value. Another common change is to specify the **Fixed-step** option in the **Solver options** panel to reflect the discrete-time nature of the DSP model.

The DSP Blockset provides an M-file, `dspstartup`, that lets you automate this configuration process so that every new model you create is preconfigured for DSP simulation. The M-file executes the following commands:

```
set_param(0, ...
    'SingleTaskRateTransMsg','error', ...
    'Solver',               'fixedstepdiscrete', ...
    'SolverMode',           'SingleTasking', ...
    'StartTime',            '0.0', ...
    'StopTime',             'inf', ...
    'FixedStep',            'auto', ...
    'SaveTime',             'off', ...
    'SaveOutput',           'off', ...
    'AlgebraicLoopMsg',     'error', ...
    'InvariantConstants',   'on', ...
    'ShowInportBlksSampModeDlgField','on', ...
    'RTWOptions',        [get_param(0,'RTWOptions')
                             ' -aRollThreshold=2']);
```

The following sections provide information about `dspstartup`:

- “Using dspstartup.m” on page C-3
- “Customizing dspstartup.m” on page C-3
- “Performance-Related Settings in dspstartup.m” on page C-4
- “Miscellaneous Settings” on page C-7

For complete information on any of the settings, see the Simulink documentation.

Using dspstartup.m

There are two ways to use the `dspstartup` M-file to preconfigure Simulink for DSP simulations:

- Run it from the MATLAB command line, by typing `dspstartup`, to preconfigure all of the models that you subsequently create. Existing models are not affected.
- Place a call to `dspstartup` within the `startup.m` file. This is an efficient way to use `dspstartup` if you would like these settings to be in effect every time you start Simulink.

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`, and save the file as `startup.m`. The result should look like something like this:

```
%STARTUP Startup file
%   This file is executed when MATLAB starts up,
%   if it exists anywhere on the path.

dspstartup;
```

The default settings in `dspstartup` will now be in effect every time you start Simulink.

For more information about performing automated tasks at startup, see the documentation for the `startup` command in the MATLAB Function Reference.

Customizing dspstartup.m

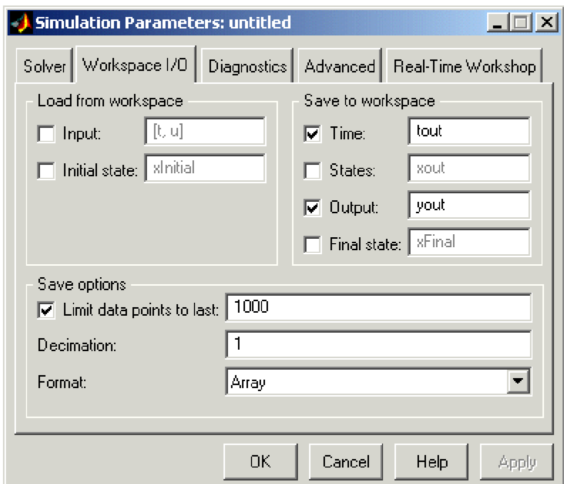
You can edit the `dspstartup` M-file to change any of the settings above or to add your own custom settings. For example, you can change the `'StopTime'` option to a value that is better suited to your particular simulations, or set the `'SaveTime'` option to `'on'` if you prefer to record the simulation sample times.

Performance-Related Settings in dspstartup.m

A number of the settings in the `dspstartup` M-file are chosen to improve the performance of the simulation:

- `'SaveTime'` is set to `'off'`

  When `'SaveTime'` is set to `'off'`, Simulink does not save the `tout` time-step vector to the workspace. The time-step record is not usually needed for analyzing discrete-time simulations, and disabling it saves a considerable amount of memory, especially when the simulation runs for an extended period of time. To enable time recording for a particular model, select the **Time** check box in the **Workspace I/O** panel of the **Simulation Parameters** dialog box (shown below).



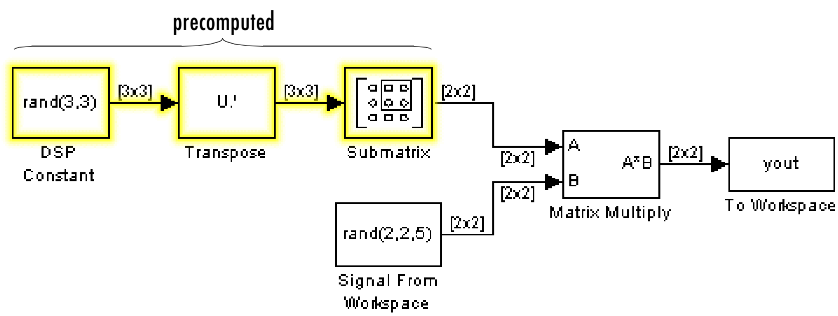
- `'SaveOutput'` is set to `'off'`

  When `'SaveOutput'` is set to `'off'`, Simulink Outport blocks in the top level of a model do not generate an output (`yout`) in the workspace. To reenable output recording for a particular model, select the **Output** check box in the **Workspace I/O** panel of the **Simulation Parameters** dialog box (above).

- `'InvariantConstants'` is set to `'on'`

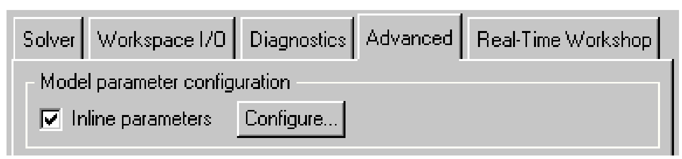
  When `'InvariantConstants'` is set to `'on'`, Simulink precomputes the values of all constant blocks (e.g., DSP Constant, Constant Diagonal Matrix) at the start of the simulation, and does not update them again for the duration of the simulation. Simulink additionally precomputes the outputs of all downstream blocks driven exclusively by constant blocks.

In the example below, the input to the top port (`U`) of the Matrix Multiply block is computed only once, at the start of the simulation.



This eliminates the computational overhead of continuously reevaluating these constant branches, which in turn results in faster simulation, and smaller and more efficient generated code.

Note, however, that when `'InvariantConstants'` is set to `'on'`, changes that you make to parameters in a constant block while the simulation is running are not registered by Simulink, and do not affect the simulation. If you would like to adjust the model constants while the simulation is running, you can turn off `'InvariantConstants'` by clearing the **Inline parameters** check box in the **Advanced** panel of the **Simulation Parameters** dialog box.



- `'RTWOptions'` sets loop-rolling threshold to `2`

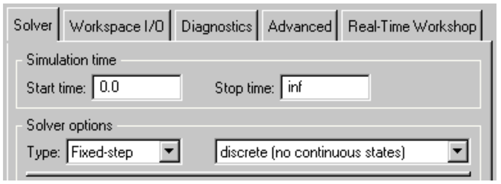
  By default, the Real-Time Workshop "unrolls" a given loop into inline code when the number of loop iterations is less than five. This avoids the overhead of servicing the loop in cases when inline code can be used with only a modest increase in the file size.

However, because typical DSP processors offer *zero-overhead looping*, code size is the primary optimization constraint in most designs. It is therefore more efficient to minimize code size by generating a loop for every instance of iteration, regardless of the number of repetitions. This is what the `'RTWOptions'` loop-rolling setting in `dspstartup` accomplishes.

Miscellaneous Settings

The `dspstartup` M-file adjusts several other parameters to make it easier to run DSP simulations. Two of the important settings are

- `'Stop time'` is set to `'Inf'`, which allows the simulation to run until you manually stop it by selecting **Stop** from the **Simulation** menu, or by clicking the **Stop Simulation** button on the toolbar. To set a finite stop time, enter a value for the **Stop time** parameter in the **Simulation Parameters** dialog box.



- `'Solver'` is set to `'fixedstepdiscrete'`, which selects the fixed-step solver option instead of the Simulink default variable-step solver. (This mode enables code generation from the model using Real-Time Workshop.) See "Discrete-Time Signals" on page 3-3 for more information about the various solver settings.

Index

Symbols

Numerics

A

N

O

P

S

T

U

Z