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)
	April 2003	Seventh printing	DSP Blockset 5.1 (Release 13 SP1)
	October 2004	Online only	DSP Blockset 5.1.1 (Release 13 SP2)

Contents

1 Introduction

2 Working with Signals

3 Filters

4 Transforms

5 Statistics, Estimation, and Linear Algebra

6 Fixed-Point Support

7 Block Reference

8 Function Reference

A Data Type Support

B Code Generation Support

C Configuring Simulink for DSP Systems

D Glossary of Fixed-Point Terms

1

Introduction

The DSP Blockset is a tool for digital signal processing (DSP) algorithm simulation and code generation. It allows you to design and prototype DSP systems by providing key DSP algorithms and components in the adaptable block format of Simulink®. This chapter provides an overview of the contents and features of the DSP Blockset.

What Is the DSP Blockset?

Welcome to the DSP Blockset, the premier tool for digital signal processing (DSP) algorithm simulation and code generation.

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.

The DSP Blockset is dependent upon Simulink, an environment for simulating dynamic systems. If you have never used Simulink before, take some time to get acquainted with its features. Simulink is a *model definition* environment. You define a model by creating a block diagram that represents the computations of your system or application. Simulink is also a *model simulation* environment. You can see how your system behaves by simulating the block diagram of your system. All of the blocks in the DSP Blockset are designed for use together with the blocks in the Simulink libraries.

If you are new to Simulink, read the Getting Started section of the documentation in order to better understand the basic functionality. You can also browse through the Simulink documentation to get complete exposure to all of the capabilities of Simulink.

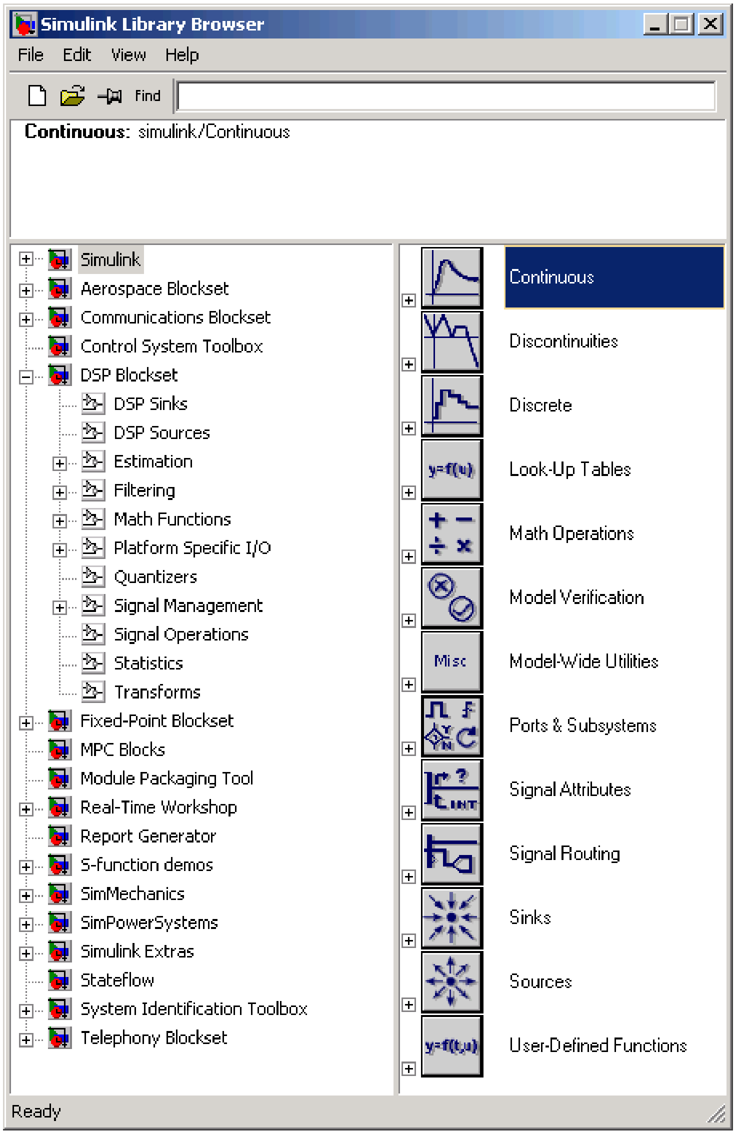
You can use the DSP Blockset and Simulink to develop your DSP concepts, and to efficiently revise and test until your design is production-ready. You can also use the DSP Blockset together with the Real-Time Workshop® to automatically generate code for real-time execution on DSP hardware.

This section includes the following topics:

- "DSP Blockset Blocks" on page 1-2 — Overview of the available blocks
- "Installing the DSP Blockset" on page 1-5 — Information on installing the DSP Blockset

DSP Blockset Blocks

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.

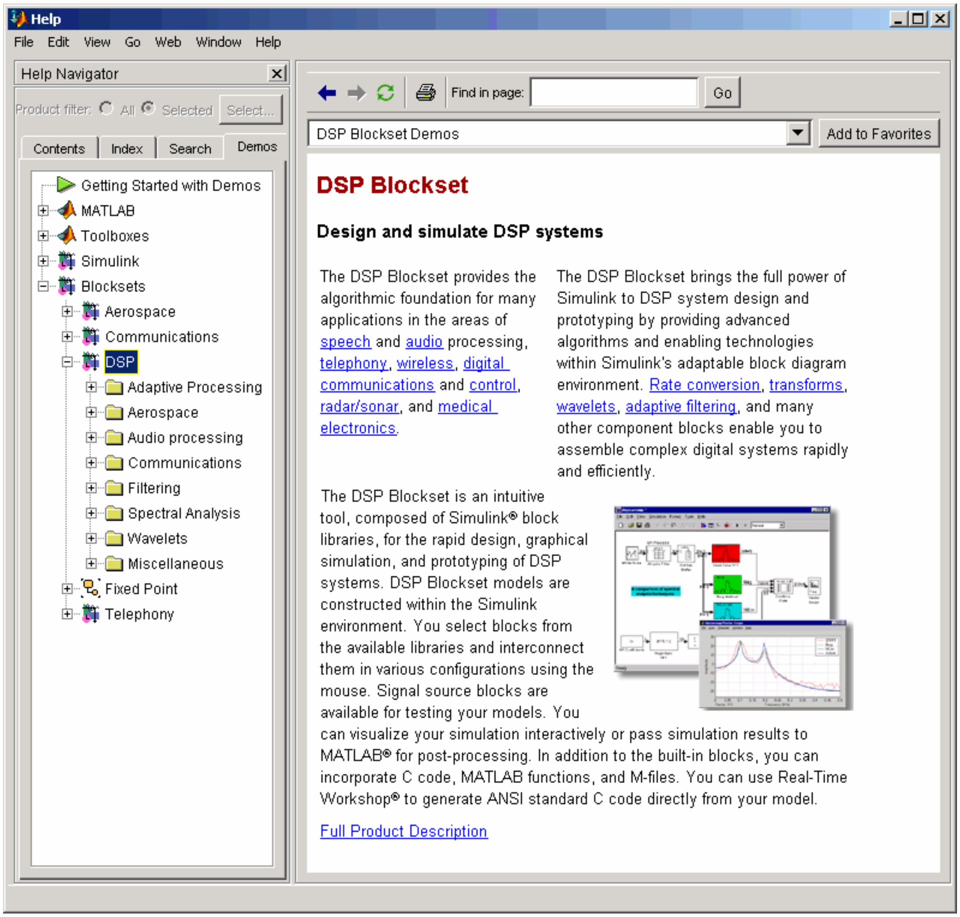


Continuous: simulink/Continuous

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.



DSP Blockset

Design and simulate DSP systems

The DSP Blockset provides the algorithmic foundation for many applications in the areas of speech and audio processing, telephony, wireless, digital communications and control, radar/sonar, and medical electronics.

The DSP Blockset brings the full power of Simulink to DSP system design and prototyping by providing advanced algorithms and enabling technologies within Simulink's adaptable block diagram environment. Rate conversion, transforms, wavelets, adaptive filtering, and many other component blocks enable you to assemble complex digital systems rapidly and efficiently.

The DSP Blockset is an intuitive tool, composed of Simulink® block libraries, for the rapid design, graphical simulation, and prototyping of DSP systems. DSP Blockset models are constructed within the Simulink environment. You select blocks from the available libraries and interconnect them in various configurations using the mouse. Signal source blocks are available for testing your models. You can visualize your simulation interactively or pass simulation results to MATLAB® for post-processing. In addition to the built-in blocks, you can incorporate C code, MATLAB functions, and M-files. You can use Real-Time Workshop® to generate ANSI standard C code directly from your model.

Full Product Description

Click on a demo name highlighted on the page to open the model. 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 Chapter 7, "Block Reference."

Installing the DSP Blockset

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

Features of the DSP Blockset

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

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, by combining them with blocks from other libraries to create sophisticated DSP systems.

This section includes the following topics:

- "Frame-Based Operations" on page 1-6 — Perform frame-based operations to improve performance.
- "Matrix Support" on page 1-7 — Represent multichannel frame-based signals using matrices
- "Data Type Support" on page 1-8 — Learn the data types supported by the DSP Blockset
- "Adaptive and Multirate Filtering" in Chapter 1 — Build advanced DSP models using this libraries
- "Statistical Operations" on page 1-10 — Perform basic statistical analysis on your signals
- "Linear Algebra" on page 1-10 — Solve equations and use matrix factorization methods
- "Parametric Estimation" on page 1-10 — Compute AR system parameters
- "Real-Time Code Generation" on page 1-11 — Create ANSI/ISO C code from your DSP model

Frame-Based Operations

Most real-time DSP systems optimize throughput rates by processing data in "batch" or "frame-based" mode. A 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.

For an example of frame-based operations, open the LPC Analysis and Synthesis of Speech demo (`dsplpc`). To run this demo, from the **Simulation** menu, select **Start**. The wide double lines that connect the blocks indicate that a frame-based signal is the input to this system. This frame-based signal is used for computation throughout the model.

The DSP Blockset delivers the 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 2-15 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 (such as 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 of the DSP Blockset User's Guide for more information about working with matrices:

- "Multichannel Signals" on page 2-10
- "Creating Signals" on page 2-32
- "Constructing Signals" on page 2-41
- "Importing Signals" on page 2-69

Data Type Support

All DSP Blockset blocks support single- and double-precision floating-point data types during both simulation and Real-Time Workshop C code generation. Many blocks also support fixed-point and Boolean data types. The following table lists all data types supported by the DSP Blockset and which block to use when converting between data types. To see which data types a particular block supports, check the "Supported Data Types" section of the block's reference page.

Supported Data Types

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-6.
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 (Continued)

Data Types Supported by DSP Blockset Blocks	Commands and Blocks for Converting Data Types	Comments
Fixed-point data types	• 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 data types in the DSP Blockset, see Chapter 6, “Fixed-Point Support.”
Custom data types	See “Correctly Defining Custom Data Types” on page A-5 to learn about custom data types.	

For more information, see Appendix A, “Data Type Support.”

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.

For an example of adaptive filtering, run the LMS Adaptive Equalization demo (`lmsadeq`). Equalization is important in the field of communications. It involves estimating and eliminating dispersion present in communication channels. In this demo, the LMS Filter block models the system’s dispersion. The plot of the squared error demonstrates the effectiveness of this adaptive filter.

For an example of mutirate filtering, run the Sample Rate Conversion demo (dspsrcnv). This demo demonstrates two ways in which you can interpolate, filter, and decimate a signal. You can use either the Upsample, Direct-Form II Transpose Filter, and Downsample blocks, or the FIR Rate Conversion block.

See “Multirate Filters” on page 3-61 and “Adaptive Filters” on page 3-46 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 (for example, mean, standard deviation, and so on), as well as the frequency distribution of input values (histograms).

Run the Statistical functions demo (statsdem) to view the maximum, mean, and variance of a noisy input signal.

See “Statistics” on page 5-2 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 Cholesky, LU, LDL, and QR factorizations are available.

See “Linear Algebra” on page 5-6 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.

In the Comparison of Spectral Analysis Techniques demo (dspsacomp), a Gaussian noise sample is being filtered by an IIR all-pole filter. Three different blocks, each with its own method, are estimating the spectrum of the IIR filter. You can view the results on a Vector Scope block.

Real-Time Code Generation

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

More information can be found in Appendix B, “Code Generation Support.”

Required Products

The DSP Blockset is part of a family of products from The MathWorks. You need to install the following products to use the DSP Blockset:

- "MATLAB" on page 1-12 — Open model files and view signal values in the MATLAB workspace.
- "Simulink" on page 1-12 — Build and simulate DSP models by connecting blocks in Simulink model files.
- "Signal Processing Toolbox" on page 1-12 — Add filters to your models with capabilities provided by the Filter Design and Analysis Tool (FDATool).

MATLAB

You can use MATLAB to open models files and view DSP Blockset demos. You can also import signal values into a DSP model from the MATLAB workspace and export the resulting signal values to the MATLAB workspace.

MATLAB documentation — For information on how to work with data and how to use the functions supplied with MATLAB, see the MATLAB documentation.

Simulink

Simulink provides an environment where you model your physical system as a block diagram. You create the block diagram by using a mouse to connect blocks and a keyboard to edit block parameters.

Simulink documentation — For information on how to connect blocks, build models, and change block parameters, see the Simulink documentation.

Signal Processing Toolbox

The Signal Processing Toolbox provides basic filter capabilities. You can design and implement filters using the Filter Design and Analysis Tool and use them in the DSP models.

Signal Processing Toolbox documentation — For information on how to use FDATool for rapid filter design and analysis, see the Signal Processing Toolbox documentation.

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
Embedded Target for Motorola® MPC555	Deploy production code onto the Motorola® MPC555
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
Real-Time Workshop Embedded Coder	Generate production code for embedded systems

Documentation and Help

The DSP Blockset software ships with printed documentation.This documentation is available online through the MATLAB Help browser, or as a PDF file that you can print or view online. For more detailed information, see the MATLAB Installation documentation for your platform.

This section includes the following topics:

- “Installing Online Documentation” on page 1-14 — Install HTML files from the Product or Documentation CD or from a Web download.
- “Viewing Online Documentation” on page 1-15 — View HTML files from your hard drive, the Documentation CD, or the MathWorks Web site.
- “Printing the Documentation” on page 1-16 — Locate and print PDF files on the MathWorks Web site.

Installing Online Documentation

Installing the online documentation is part of the normal MathWorks installation process:

- Documentation from a CD — If you are on a Windows platform, start the MathWorks installer. When prompted, select the **Install products and documentation** option button.

  If you are on a UNIX platform, start the installer, and select the products and documentation you want to install.
- Documentation from a Web download — If you update the DSP Blockset using a Web download, and you want to view the documentation with the MathWorks Help browser, you must install the documentation on your hard drive.

  If you are on a Windows platform, download the files from the Web. Then, start the installer, and select the **Install products and documentation** option button.

  If you are on a UNIX platform, download the files from the Web. Then, start the installer, and select the products and documentation you want to install.

Viewing Online Documentation

You can access the online documentation from HTML files you install on your hard drive, from the Documentation CD, or through the MathWorks technical support Web pages.

To Access HTML Documentation on Your Hard Drive or the Product or Documentation CD

This procedure assumes you are working on a Windows platform:

1 In the MATLAB window, from the **Help** menu, click **Full Product Family Help**.

   The Help browser window opens.

2 In the left pane, click **DSP Blockset**.

   In the right pane, the Help browser displays the DSP Blockset Roadmap page. If you did not install the HTML files on your hard drive, a message box opens asking you to insert your Documentation CD.

3 Under the section titled Required and Related Products, select Related Products.

   The Help browser displays the DSP Blockset Related Products information.

   Note If you installed the DSP Blockset from a Web download, and you chose not to install the HTML help files, the current documentation is neither on your hard drive, nor on the Documentation CD. You need to use the MathWorks technical support Web site.

To Access HTML Documentation from MathWorks Technical Support

Alternatively, you can view the documentation from the MathWorks technical support Web site. The Web pages are identical to the latest release whether it was distributed from a CD or a Web download:

1 Open a Web browser.

2 In the address box, enter

   `http://www.mathworks.com/access/helpdesk/help/toolbox/dspblks/dspblks.shtml`

   to view the DSP Blockset documentation on the Mathworks Web site.

Printing the Documentation

The documentation for the DSP Blockset is available as PDF files. You need to install Adobe Acrobat Reader 4.0 or later to open and read these files. To download a free copy of Acrobat Reader, see `http://www.adobe.com/products/acrobat/main.html`:

1 To view the documentation in PDF format, open a Web browser.

2 In the address box, enter

   `http://www.mathworks.com/access/helpdesk/help/toolbox/dspblks/dspblks.shtml`

   This is the DSP Blockset Roadmap page.

3 Under the **Printing the Documentation** heading, click on the links to view PDF versions of the documentation.

Definitions and Nomenclature

The following section contains descriptions of the terms you should understand before reading this guide.

This section includes the following topics:

- "Signals" on page 1-17 — Definitions of frame-based signals and sample-based signals.
- "Sampling" on page 1-18 — Description of sample-based signals and frame-based signals.
- "Tunable Parameters" on page 1-18 — Change the value of a block's parameter while the simulation is running.

Signals

Signals in Simulink can be real or complex valued and represented with data types such as single, double, and fixed point. Signals can be either sample based or frame 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.

Sample-Based Signals

A signal is sample based if it is propagated through the model sample-by-sample. If there is only one independent sequence of numbers, the signal is a single-channel signal. 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. Each matrix element represents one sample from a distinct channel. In order to represent multiple values from multiple channels, you must create a three dimensional matrix. The third dimension of this matrix represents the sequential values of the sample-based signal in the channels.

Frame-Based Signals

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 single frame can be propagated through the model at one time, thus improving computational time. 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.

Sampling

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*. Signals defined in this way are called sample-based signals. 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.

Note In the block dialog boxes, the term *sample time* is used to refer to the *sample period*, T_s.

A frame-based signal is composed of a series of samples grouped within one frame. The *input frame period* (T_{fi}) of a frame-based signal is the time interval between consecutive vector or matrix inputs to a block. Similarly, the *output frame period* (T_{fo}) is the time interval at which the block updates the frame-based vector or matrix value at the output port.

Tunable Parameters

There are some parameters, such as the Sine Wave block's **Frequency** parameter, that you can change or tune while a simulation runs. 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.

Note Opening a dialog box for a source block causes the simulation to pause. While the simulation is paused, you can edit the parameter values. However, you must close the dialog box to have the changes take effect and allow the simulation to continue.

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**. The simulation continues to run, but with the new parameter settings.

Tunable Parameters During Simulation

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.

Typographical Conventions

This manual 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, user input, items in drop-down lists	`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`*`')`

2

Working with Signals

This chapter helps you understand how signals are represented in Simulink. It also explains how to create, construct, import, export, and view signals.

Signal Concepts

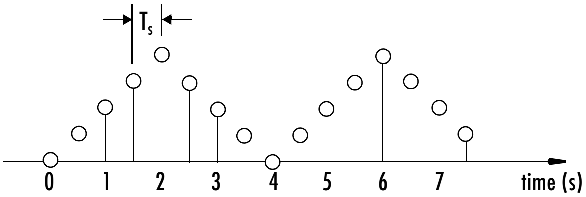
Simulink models can process both discrete-time and continuous-time signals, although models that are built with the DSP Blockset are often intended to process only discrete-time signals. The next few sections cover the following topics:

- "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 2-8 — An explanation of how continuous-time signals are treated by various blocks in the DSP Blockset
- "Multichannel Signals" on page 2-10 — A description of how multichannel signals are represented in Simulink
- "Benefits of Frame-Based Processing" on page 2-13 — 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 2-4

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.

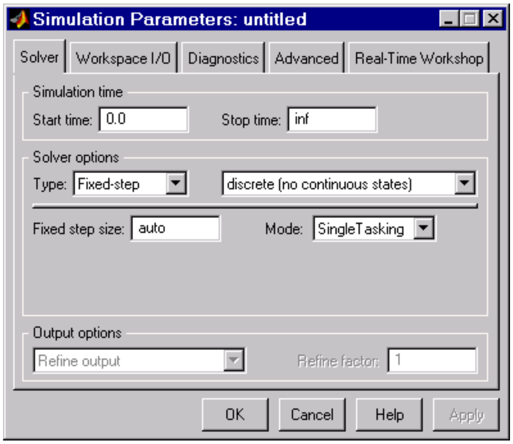
Term	Symbol	Units	Notes
Sample period	T_s T_{si} T_{so},	Seconds	The time interval between consecutive samples in a sequence, as the input to a block (T_{si}) or the output from a block (T_{so}).
Frame period	T_f T_{fi} T_{fo}	Seconds	The time interval between consecutive frames in a sequence, as the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



Simulation time
Start time: 0.0 Stop time: inf

Solver options
Type: Fixed-step discrete (no continuous states)
Fixed step size: auto Mode: SingleTasking

Output options
Refine output Refine factor: 1

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

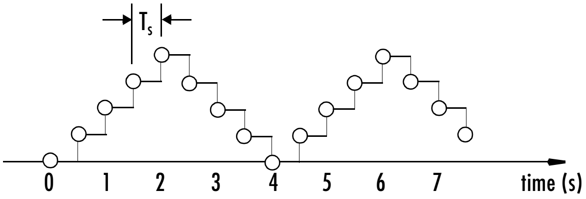
- "Recommended Settings for Discrete-Time Simulations"
- "Additional Settings for Discrete-Time Simulations" on page 2-6
- "Cross-Rate Operations in Variable-Step and Fixed-Step SingleTasking Modes" on page 2-6
- "Sample Time Offsets" on page 2-8

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

- **Type**: `Fixed-step discrete (no continuous states)`
- **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 2-2 by remaining *defined* between sample times. For example, the representation of the discrete-time triangle wave looks like this.



The above signal's value at t=3.112 seconds is the same as the signal's value at t=3 seconds. In `Fixed-step SingleTasking` mode, a signal's sample times are the instants where the signal is allowed to *change* values, rather than where the signal is defined. Between the sample times, the signal takes on the value at the previous sample time.

As a result, in `Fixed-step SingleTasking` mode, Simulink permits cross-rate operations such as the addition of two signals of different rates. This is explained further in "Cross-Rate Operations in Variable-Step and Fixed-Step SingleTasking Modes" on page 2-6.

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 2-2. 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 2-19 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 2-5. In this mode, cross-rate operations are allowed by Simulink.

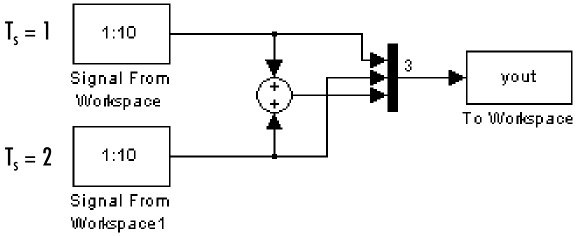
In the `Fixed-step Auto` setting, Simulink automatically selects a tasking mode (single-tasking or multitasking) that is best suited to the model. See “Simulink Tasking Mode” on page 2-98 for a description of the criteria that Simulink uses to make this decision. For the typical model containing multiple 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:

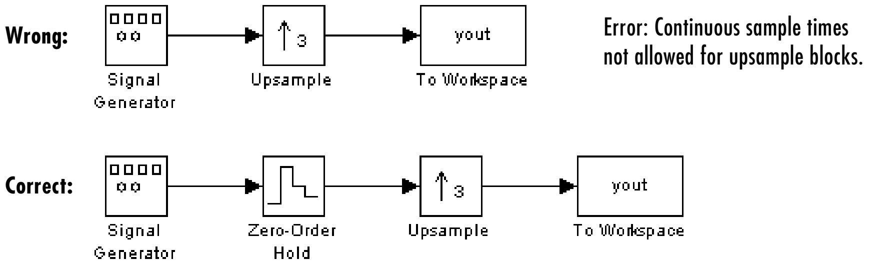
- "Source Blocks"
- "Nonsource Blocks" on page 2-9

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 2-69 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.



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

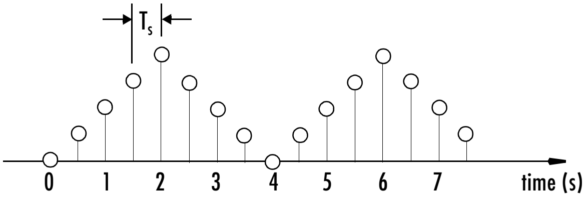
All nonsource blocks in the DSP Blockset accept discrete signals, and inherit the sample period of the input. Others additionally accept continuous-time discrete signals.

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 2-2. 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:

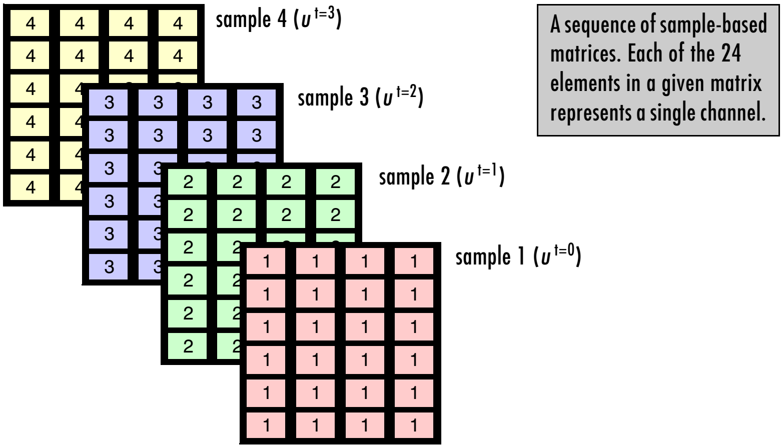
- "Sample-Based Multichannel Signals"
- "Frame-Based Multichannel Signals" on page 2-11

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.



A sequence of sample-based matrices. Each of the 24 elements in a given matrix represents a single channel.

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:

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

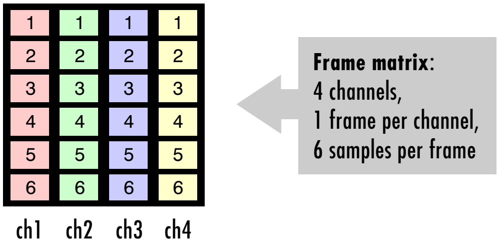
See these sections for information about working with sample-based multichannel signals:

- “Creating Signals” on page 2-32
- “Constructing Signals” on page 2-41
- “Deconstructing Signals” on page 2-61
- “Importing Signals” on page 2-69
- “Exporting Signals” on page 2-79
- “Viewing Signals” on page 2-87

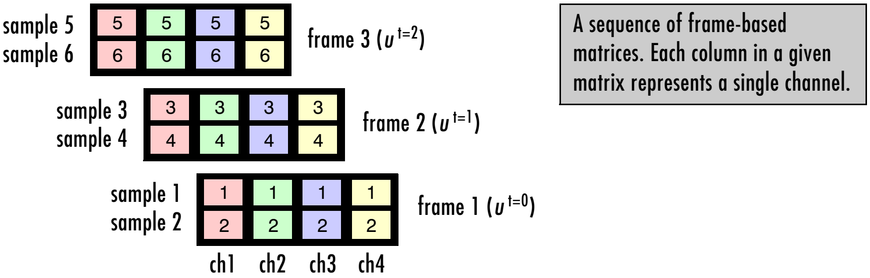
Frame-Based Multichannel Signals

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 these sections for information about working with frame-based multichannel signals:

- "Creating Signals" on page 2-32
- "Constructing Signals" on page 2-41
- "Deconstructing Signals" on page 2-61
- "Importing Signals" on page 2-69
- "Exporting Signals" on page 2-79
- "Viewing Signals" on page 2-87

Benefits of Frame-Based Processing

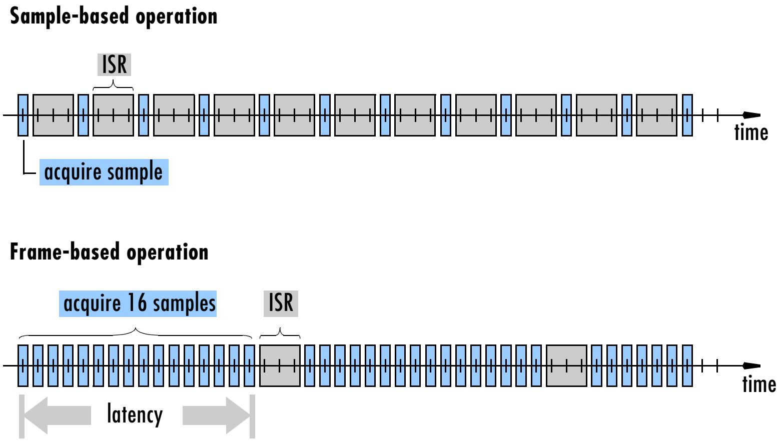
Frame-based processing is an established method of accelerating both real-time systems and simulations.

Accelerating Real-Time Systems

Frame-based data is a common format in real-time systems. Data acquisition hardware often operates by accumulating a large number of signal samples at a high rate, and propagating these samples to the real-time system as a block of data. This maximizes the efficiency of the system by distributing the fixed process overhead across many samples; the "fast" data acquisition is suspended by "slow" interrupt processes after each frame is acquired, rather than after each individual sample.

The figure below illustrates how throughput is increased by frame-based data acquisition. The thin blocks each represent the time elapsed during acquisition of a sample. The thicker blocks each represent the time elapsed during the interrupt service routine (ISR) that reads the data from the hardware.

In this example, the frame-based operation acquires a frame of 16 samples between each ISR. The frame-based throughput rate is therefore many times higher than the sample-based alternative.



It's important to note that frame-based processing 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 2-16
- “Converting Sample Rates and Frame Rates” on page 2-19
- “Changing Frame Status” on page 2-30

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 2-16 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:

$$T_s = T_f/M$$

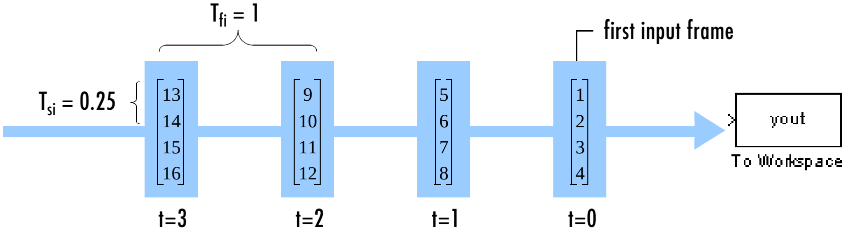
More specifically, the sample periods of inputs (T_{si}) and outputs (T_{so}) are related to their respective frame periods by

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

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

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 2-19.

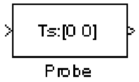
Inspecting Sample Rates and Frame Rates

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 2-18

Using the Probe Block to Inspect Rates

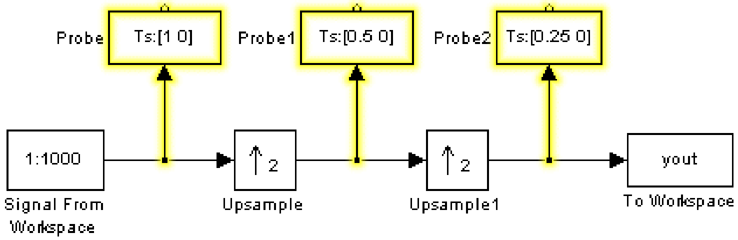
Connect the Simulink Probe block to any line to display the period of the signal on that line. The period is displayed in the block icon itself (together with the line width and data type, if desired), making it easy to verify that the sample rates in the model are what you expect them to be. When the 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.



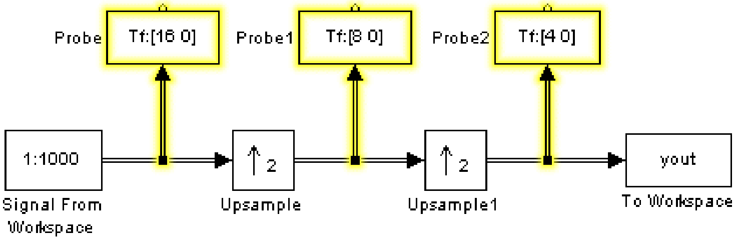
The block displays the label `Ts` or `Tf`, followed by a two-element vector. The first (left) element is the period of the signal being measured. The second (right) is the signal's sample time offset, which is usually `0`, as explained in "Sample Time Offsets" on page 2-8.

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 2-15.

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.

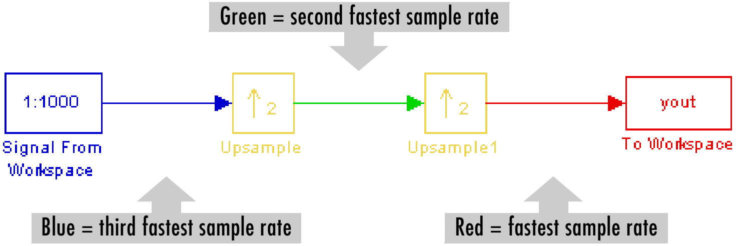


Note that the sample rate conversion is implemented through a change in the frame period rather than the frame size. This is because the **Frame-based mode** parameter in the Upsample blocks is set to `Maintain input frame size` rather than `Maintain input frame rate`. See “Converting Sample Rates and Frame Rates” on page 2-19 for more information.

Using Sample Time Color Coding to Inspect Sample Rates

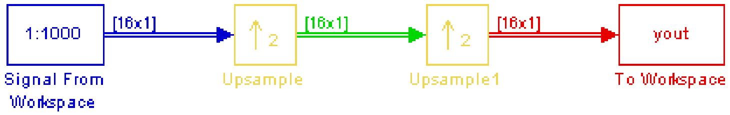
Turn on the sample time color coding option in Simulink by selecting **Sample time colors** from the **Format** menu. For sample-based signals, this assigns each sample rate a different color. For frame-based signals, this assigns each frame rate a different color.

Sample Time Color Coding Example: Sample-Based. Here is the sample-based model from “Probe Block Example: Sample-Based” on page 2-17 with the Probe blocks removed and sample time color coding turned on.


Green = second fastest sample rate
1:1000
Signal From Workspace
↑2
Upsample
↑2
Upsample1
yout
To Workspace
Blue = third fastest sample rate
Red = fastest sample rate

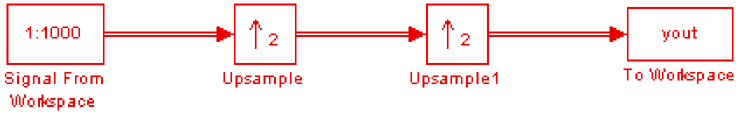
Since every sample-based signal in this model has a different sample rate, each signal is assigned a different color.

Sample Time Color Coding Example: Frame-Based. Here’s the frame-based model from “Probe Block Example: Frame-Based” on page 2-17 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 2-23
- "Avoiding Unintended Rate Conversions" on page 2-27

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 2-92 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 (that is $M_o \neq M_i$), but keep the frame rate constant ($T_{fo} = T_{fi}$)

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

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

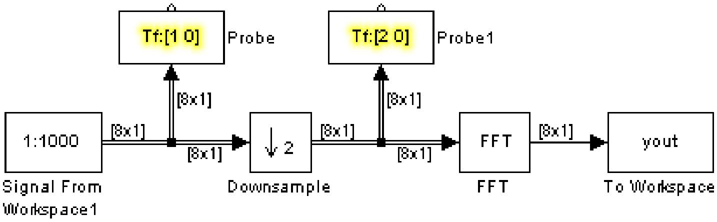
Rate Conversion Blocks. The following table lists the principal rate conversion blocks in the DSP Blockset. Blocks marked with an asterisk (*) offer the option of changing the rate by either adjusting the frame size or frame rate.

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

The following examples illustrate the two sample rate conversion modes:

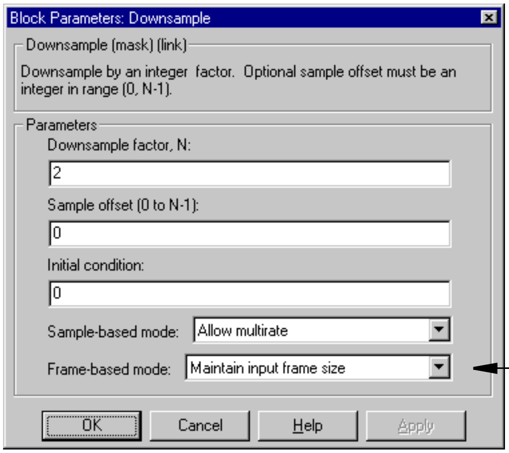
- "Example: Rate Conversion by Frame-Rate Adjustment"
- "Example: Rate Conversion by Frame-Size Adjustment" on page 2-22

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.


Block Parameters: Downsample

Downsample (mask) (link)

Downsample by an integer factor. Optional sample offset must be an integer in range [0, N-1].

Parameters

Downsample factor, N: 2

Sample offset (0 to N-1): 0

Initial condition: 0

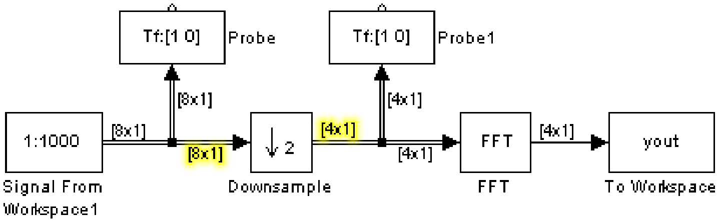
Sample-based mode: Allow multirate

Frame-based mode: Maintain input frame size

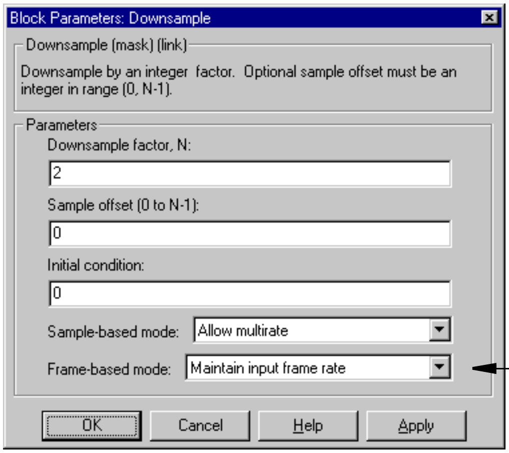
Maintain input frame size: Downsample the signal by changing the frame *rate*.

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 (that is, 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 2-21 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.


Block Parameters: Downsample

Downsample (mask) (link)

Downsample by an integer factor. Optional sample offset must be an integer in range [0, N-1].

Parameters

Downsample factor, N: 2

Sample offset (0 to N-1): 0

Initial condition: 0

Sample-based mode: Allow multirate

Frame-based mode: Maintain input frame rate

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 2-25

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 **Output buffer size (per channel)** parameter.

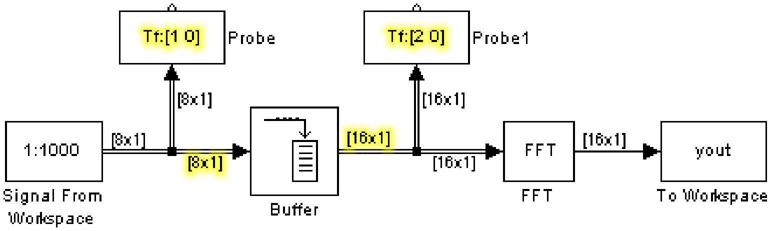
The Unbuffer block is specialized for completely unbuffering a frame-based signal to its sample-based equivalent, and always preserves the signal's data and sample period:

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

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 **Output buffer size (per channel)** parameter. As a result, the new output sample period is

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

- 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.

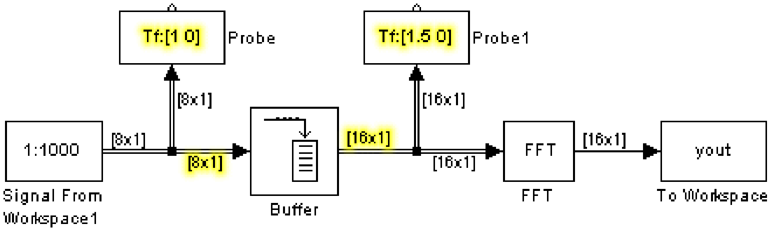
- Zero Pad adds samples to the sequence by appending zeros to each frame when **Pad along** is set to `Columns`. The output and input frame periods are the same, $T_{fo} = T_{fi}$, and the new output sample period is

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

  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

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

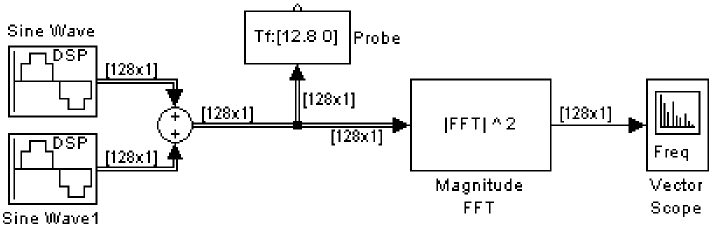
which indicates that T_{fo} should be (16-4)∗0.125, or 1.5 seconds, as confirmed by the second Probe block. The sample period of the signal at the output of the Buffer block is no longer 0.125 second, but rather 0.0938 second (that is, 1.5/16). Thus, both the signal's data and the signal's sample period 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 2-29

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

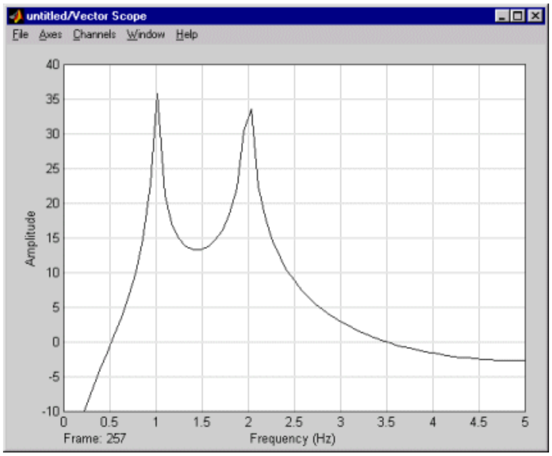
The frame period of the resulting summed sinusoid is 12.8 seconds (that is, 128∗0.1), which is confirmed by the Probe block when the model is updated.

Select the **Inherit FFT length from input dimensions** check box 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.

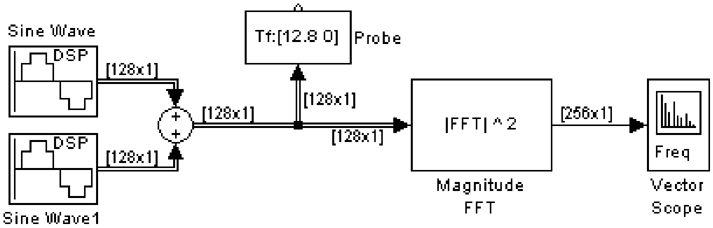


The Vector Scope block uses the input frame size (128) and period (12.8) to deduce the original signal's sample period (0.1), which allows it to *correctly* display the peaks at 1 Hz and 2 Hz.

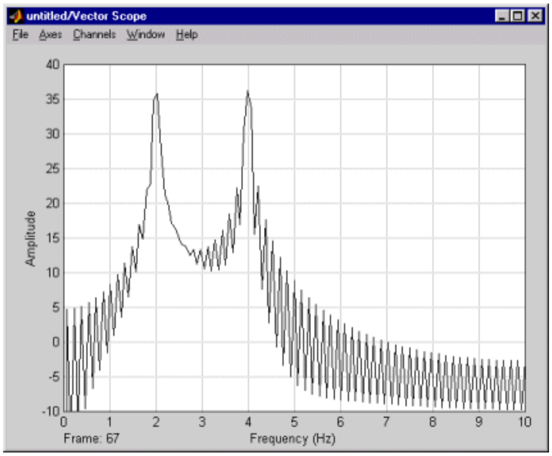
Example 2: Unintended Rate Conversion. Now alter the previous example by setting the Magnitude FFT block parameters as follows:

- 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.

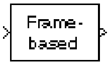
Changing Frame Status

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, →, for a sample-based signal and a double line, ⇒, 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 these sections for more information about these methods:

- “Buffering Sample-Based and Frame-Based Signals” on page 2-47
- “Unbuffering a Frame-Based Signal into a Sample-Based Signal” on page 2-67

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 2-32
- "Creating Signals Using Signal Generator Blocks" on page 2-35
- "Creating Signals Using the Signal From Workspace Block" on page 2-37

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 these sections:

- "Constructing Multichannel Sample-Based Signals" on page 2-41
- "Constructing Multichannel Frame-Based Signals" on page 2-44

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
- 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 7, "Block Reference" for a complete explanation of all the constant blocks.

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

- "Creating Signals Using Signal Generator Blocks" on page 2-35
- "Creating Signals Using the Signal From Workspace Block" on page 2-37

For information about importing signals, see these sections:

- "Importing a Multichannel Sample-Based Signal" on page 2-69
- "Importing a Multichannel Frame-Based Signal" on page 2-75

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**

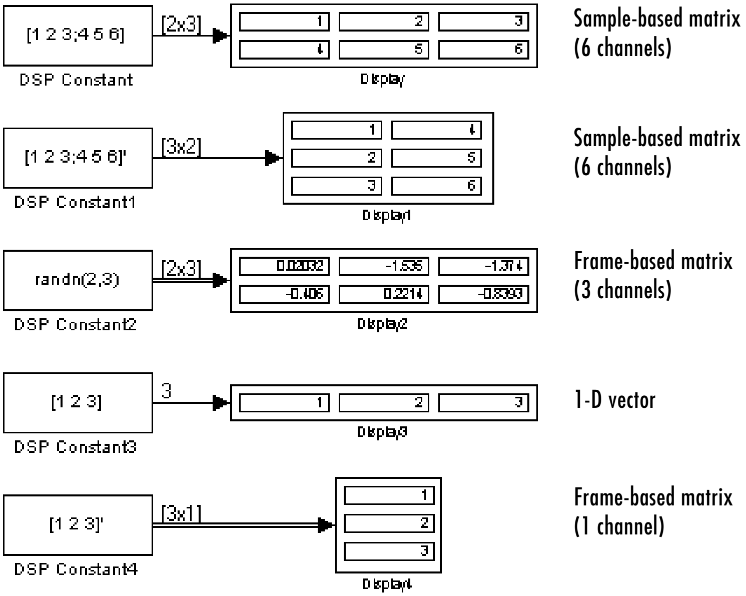
To generate a constant matrix signal, simply enter the desired matrix in the **Constant value** parameter using standard MATLAB notation. Some common examples of the MATLAB matrix notation are shown below:

```
[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 2-10 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
- 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 7, “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 these sections:

- "Creating Signals Using Constant Blocks" on page 2-32
- "Creating Signals Using the Signal From Workspace Block" on page 2-37

Example: Creating Signals with the Sine Wave Block

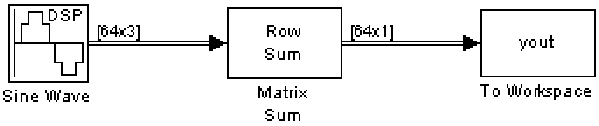
The Sine Wave block dialog box contains the following key parameters.

- **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.



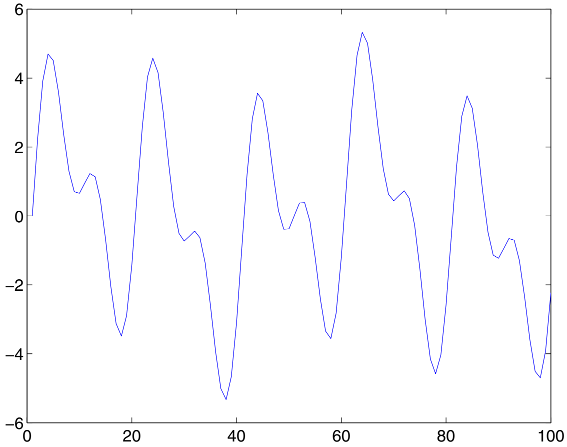
To build the model, set the **Sum along** parameter of the Matrix Sum block to **Rows**, and make the following parameter settings in the Sine Wave block:

- 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 2-10 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.

1:10
Signal From
Workspace

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 2-38
- “Example 2: Generating Frame-Based Output” on page 2-39

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

- "Importing a Multichannel Sample-Based Signal" on page 2-69
- "Importing a Multichannel Frame-Based Signal" on page 2-75

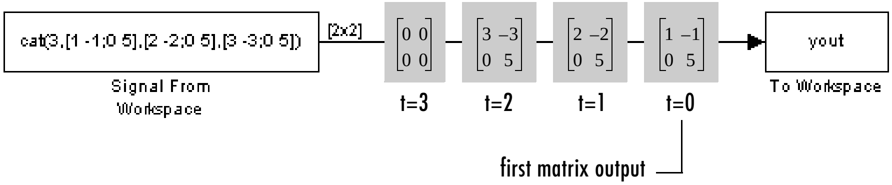
For more information about creating signals, see these sections:

- "Creating Signals Using Constant Blocks" on page 2-32
- "Creating Signals Using Signal Generator Blocks" on page 2-35

Example 1: Generating Sample-Based Output

In the model below, the Signal From Workspace block 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`

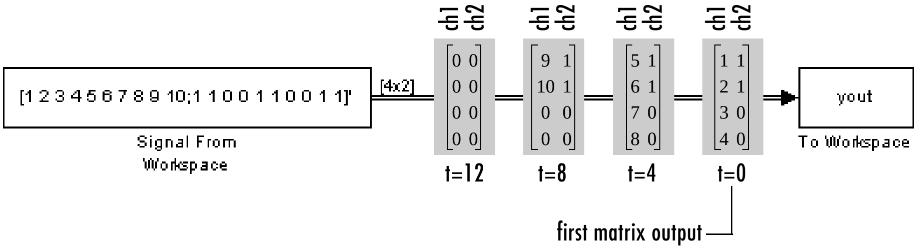
The **Sample time** setting of `1` yields a sample-based output with sample period of 1 second. Each of the four elements in the matrix signal represents an independent channel (the channel numbering is arbitrary). The **Form output**

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 2-41
- "Constructing Multichannel Frame-Based Signals" on page 2-44

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

- "Creating Signals Using Constant Blocks" on page 2-32
- "Creating Signals Using Signal Generator Blocks" on page 2-35
- "Creating Signals Using the Signal From Workspace Block" on page 2-37

Constructing Multichannel Sample-Based Signals

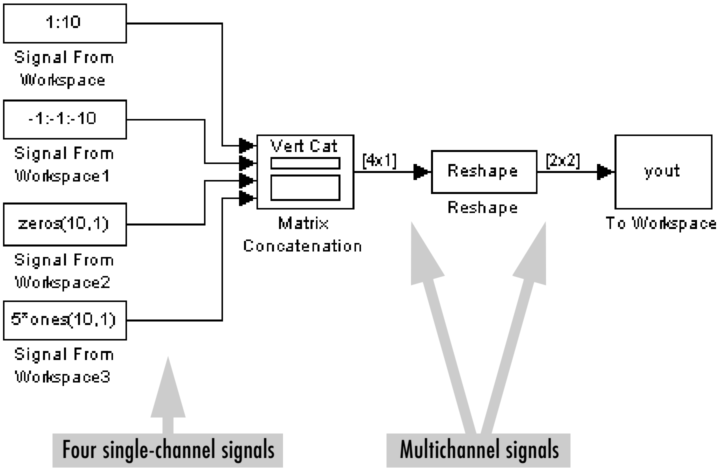
A sample-based signal with M∗N channels is represented by a sequence of M-by-N matrices. (The special case of M = N = 1 represents a single-channel signal.) Multiple individual signals can be combined into a multichannel matrix signal using the Matrix Concatenation block. Individual signals can be added 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 2-42
- "Constructing Sample-Based Multichannel Signals from Existing Sample-Based Multichannel Signals" on page 2-43

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.



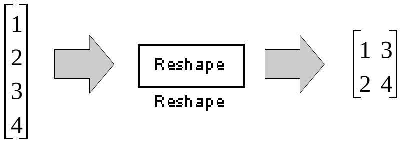
Four single-channel signals

Multichannel signals

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.

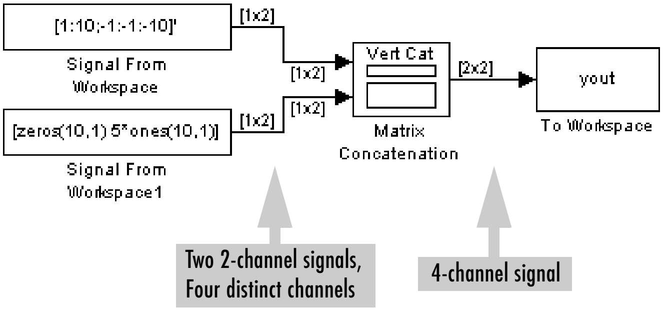


The 4-by-1 matrix and the 2-by-2 matrix in the above model represent the same sample-based four-channel signal. In some cases one representation may be more useful than the other. For more information about the Reshape block, see the Reshape block reference page. See “Sample-Based Multichannel Signals” on page 2-10 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.



Two 2-channel signals, Four distinct channels

4-channel signal

To build the model, make the following parameter settings:

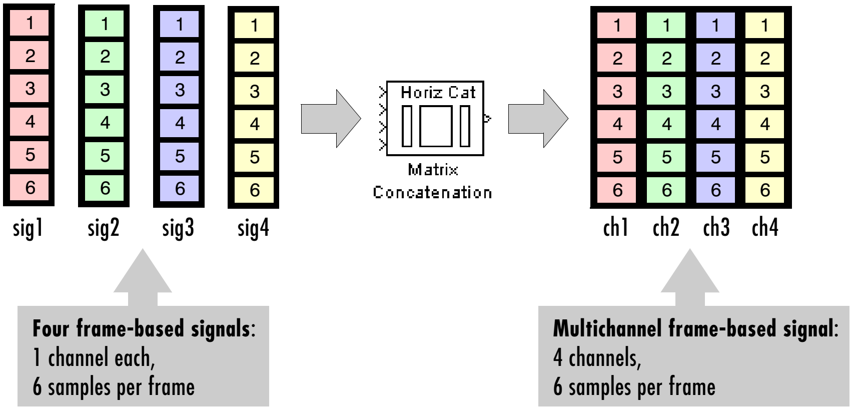
- In Signal From Workspace, set **Signal** = `[1:10;-1:-1:-10]'`
- In Signal From Workspace1, set **Signal** = `[zeros(10,1) 5*ones(10,1)]`
- In Matrix Concatenation, set:
  - **Number of inputs** = `2`
  - **Concatenation method** = `Vertical`

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 2-10 for more about sample-based signals.

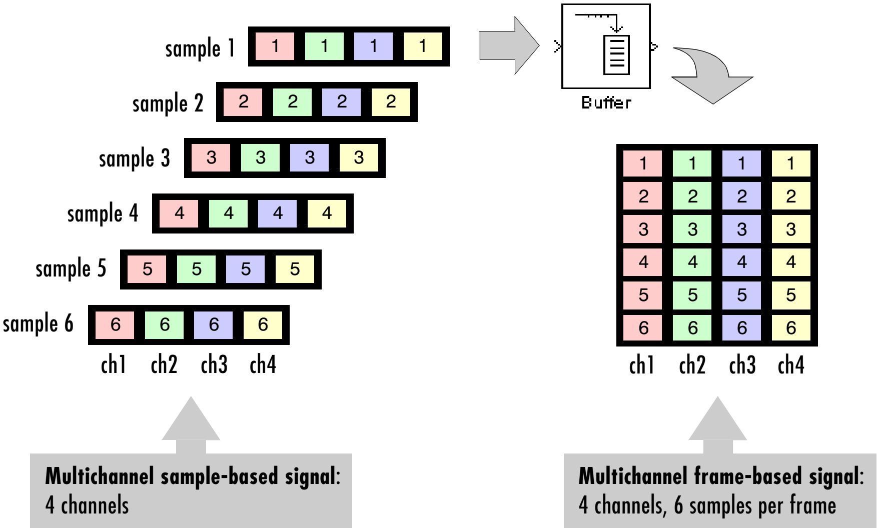
Constructing Multichannel Frame-Based Signals

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:

- *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.


1 2 3 4 5 6
sig1
sig2
sig3
sig4
Horiz Cat
Matrix Concatenation
ch1
ch2
ch3
ch4
Four frame-based signals: 1 channel each, 6 samples per frame
Multichannel frame-based signal: 4 channels, 6 samples per frame

- *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).


sample 1
sample 2
sample 3
sample 4
sample 5
sample 6
ch1
ch2
ch3
ch4
Buffer
ch1
ch2
ch3
ch4
Multichannel sample-based signal:
4 channels
Multichannel frame-based signal:
4 channels, 6 samples per frame

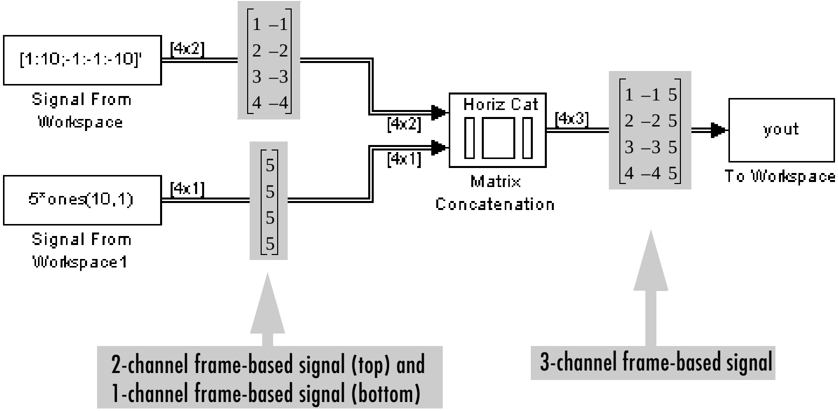
The following sections explain the two methods of constructing multichannel frame-based signals:

- "Concatenating Independent Frame-Based Signals into Multichannel Signals" on page 2-45
- "Buffering Sample-Based and Frame-Based Signals" on page 2-47

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 2-11 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 these sections for more information:

- “Creating Signals Using the Signal From Workspace Block” on page 2-37
- “Importing a Multichannel Frame-Based Signal” on page 2-75

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 **Output 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

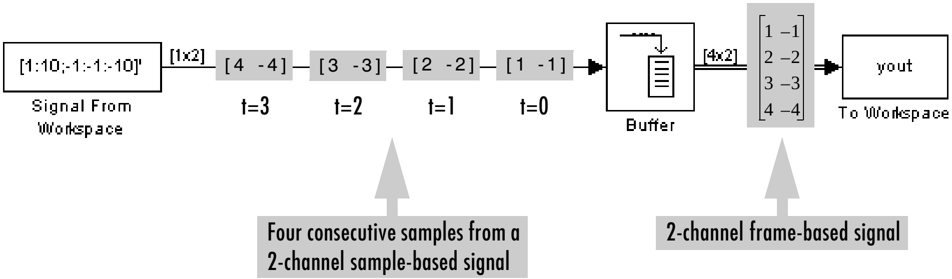
$$T_{so} = \frac{(M_o - L)T_{si}}{M_o}$$

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 2-19 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 2-48
- “Overlapping Buffers” on page 2-49
- “Example: Buffering Sample-Based Signals with Overlap” on page 2-49
- “Example: Buffering Frame-Based Signals with Overlap” on page 2-51
- “Buffering Delay and Initial Conditions” on page 2-52

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.



Four consecutive samples from a 2-channel sample-based signal

2-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 these sections for more information:

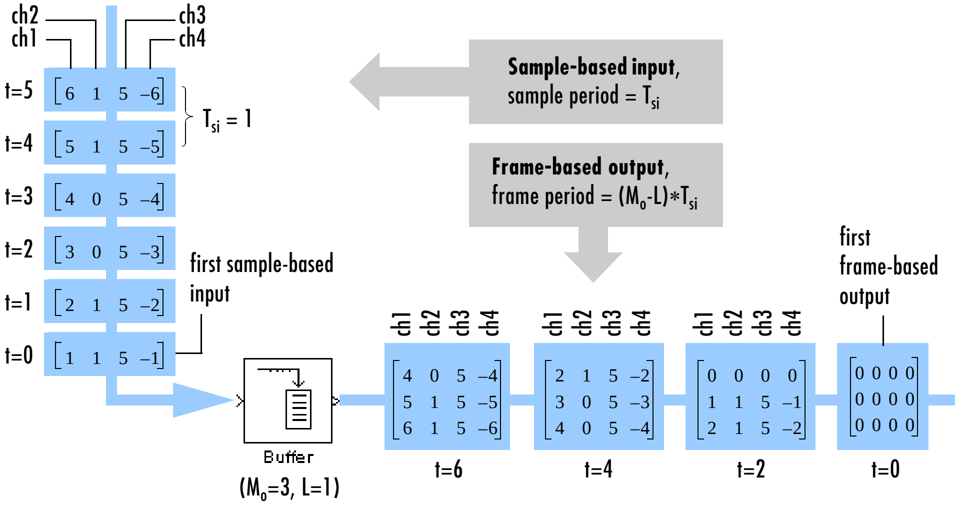
- "Creating Signals Using the Signal From Workspace Block" on page 2-37
- "Importing a Multichannel Frame-Based Signal" on page 2-75

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.

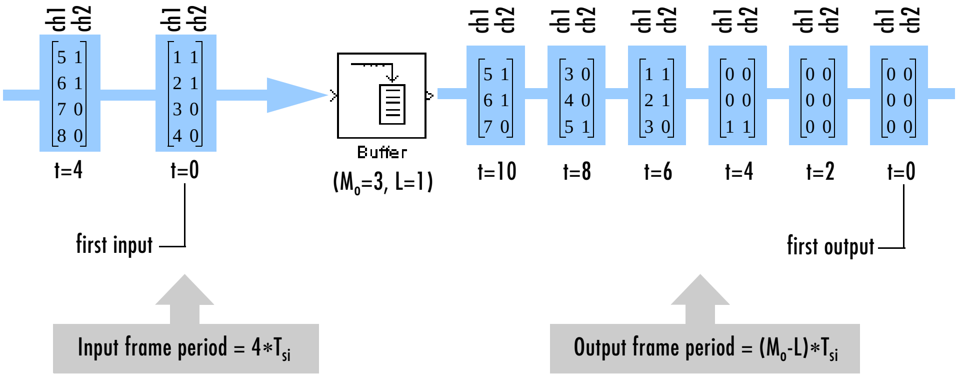
```
A = [1 1 5 -1;2 1 5 -2;3 0 5 -3;4 0 5 -4;5 1 5 -5;6 1 5 -6];
```

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 2-92 for general information about algorithmic delay, and see "Buffering Delay and Initial Conditions" on page 2-52 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 2-92 for general information about algorithmic delay, and see “Buffering Delay and Initial Conditions” on page 2-52 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” on page 2-49 the delay is four samples. In “Example: Buffering Frame-Based Signals with Overlap” on page 2-51” 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.

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
```

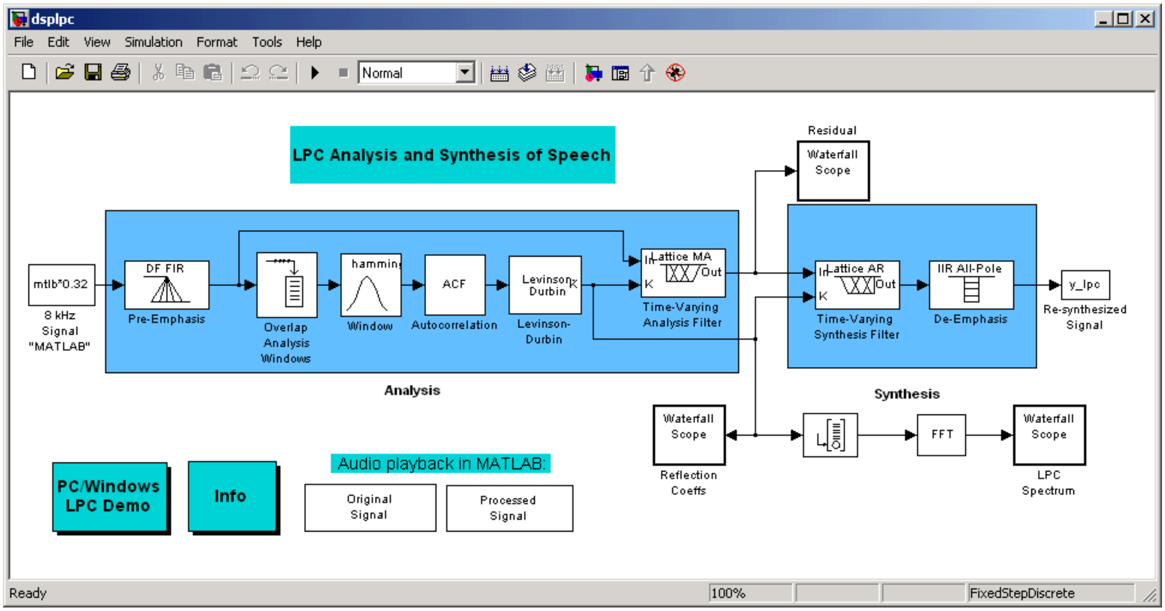
This agrees with the block’s output in that example. See “Delay and Latency” on page 2-92 and the “Latency” section on each block reference page for more information.

Analysis and Synthesis of Speech

In modern digital systems, a speech signal is represented in a digital format that is comprised of a sequence of binary bits. For storage and transmission applications, it is often desirable for the speech signal to be represented in as few bits as possible, while maintaining its perceptual quality.

In narrowband digital speech compression, digital speech signals are sampled at a rate of 8000 samples per second. Typically, each sample is represented by 8-bits. This corresponds to a bit rate of 64 kbits per second. Further compression is possible at the cost of quality. Most of the current low bit rate speech coders are based on the principle of linear predictive speech coding. The simplest implementation of this compression technique is presented in the linear prediction coefficient (LPC) Analysis and Synthesis of Speech demo. This topic describes this demo, which models the theory behind signal transmission:

1 Open the LPC Analysis and Synthesis of Speech demo by typing `dsplpc` at the MATLAB command line.



The input to this model is a human speech segment that is 0.5 second long. This model calculates the reflection coefficients of the speech segment and uses them to create the linear prediction analysis filter (lattice-structure). The model calculates the residual signal by filtering the preemphasized speech samples. The residual signal, which is the output of the analysis stage, is the low energy equivalent of the input signal and is easier to quantize. The blocks in the synthesis stage of the model use the known residual and reflection coefficients to synthesize the original signal.

2 Simulate this model.

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

There is almost no difference between the two. This is because no quantization is used.

Suppose that you want to create a model that more accurately portrays what happens during cellular phone communication. A better approximation of a real-world system would involve the quantization of the residual and reflection coefficients before they are transmitted. For information on how to design a scalar quantizer to accomplish such a task, see “Creating a Scalar Quantizer” on page 2-54.

Creating a Scalar Quantizer

In the previous topic, “Analysis and Synthesis of Speech” on page 2-52, you learned the theory behind the LPC Analysis and Synthesis of Speech (`dsplpc`) demo. In this topic, you create two scalar quantizers and add them to this demo model. One scalar quantizer is capable of quantizing the residual that is the output of the Time-Varying Analysis Filter block. The other scalar quantizer quantizes the reflection coefficients that are the output of the Levinson-Durbin block:

1 Open the LPC Analysis and Synthesis of Speech demo by typing `dsplpc` at the MATLAB command line.

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

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

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

5 Double-click this Signal To Workspace block. For the **Variable name** parameter, enter `K`. Click **OK**.

6 Connect the output of the Time-Varying Analysis Filter block to the other Signal To Workspace block.

7 Double-click this Signal To Workspace block. For the **Variable name** parameter, enter `E`. Click **OK**.

8 Run the simulation. The variables K and E are now defined in the MATLAB workspace.

9 From the Quantizers library, click-and-drag a Scalar Quantizer Design block into your model.

10 Double-click this block.

The SQ Design Tool GUI opens.

11 For the **Training Set** parameter, enter `K`. The variable K represents the reflection coefficients you want to quantize.

Note Theoretically, the signal that is used as the **Training Set** parameter should contain all the possible combinations of the parameter to be quantized. However, this example provides an approximation to a global training process.

By definition, your reflection coefficients range from -1 to 1. Assume also that your cellular phone has 7 bits to represent each reflection coefficient. This means it is capable of representing 2^7 or 128 values.

12 For the **Number of levels** parameter, enter `128`. This number is equal to the total number of codebook values.

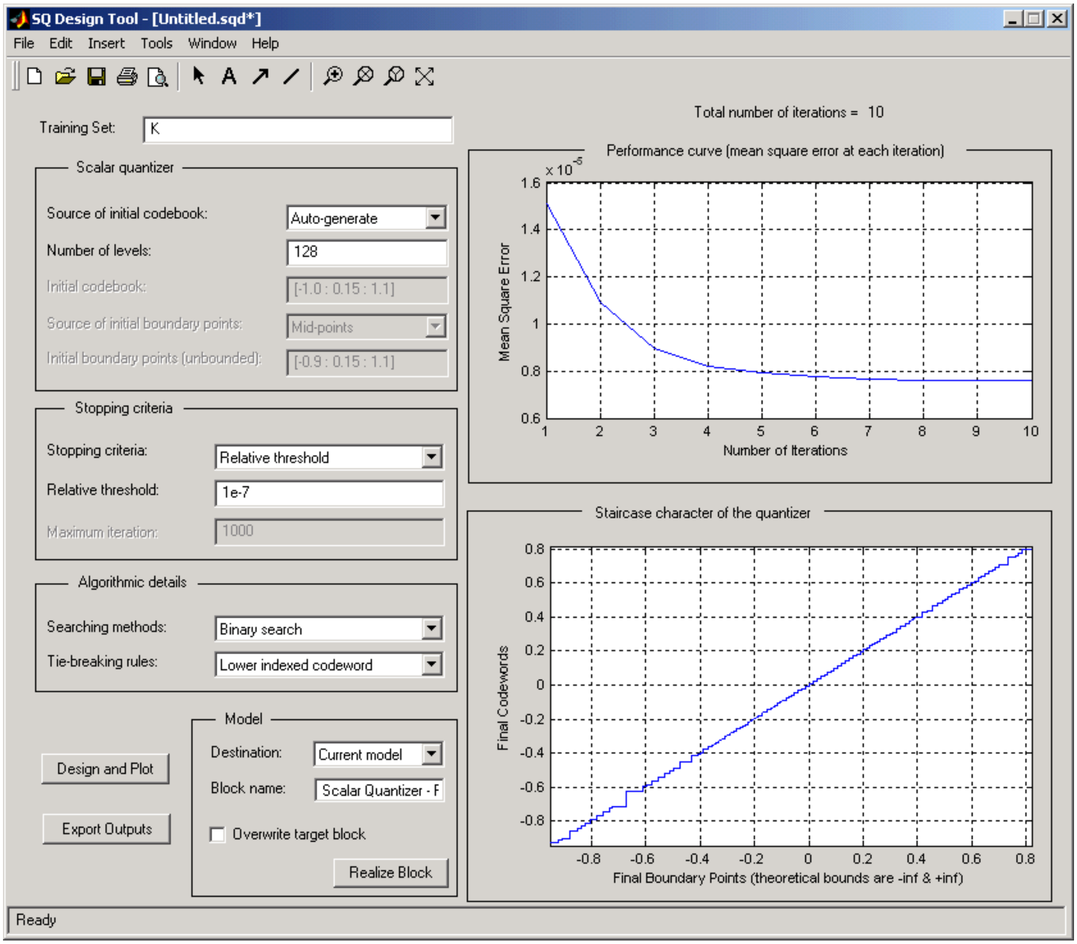
13 For the **Block name** parameter, enter `Scalar Quantizer - Reflection Coefficients`.

Leave the rest of the parameters at their default values.

14 Make sure that your desired destination model, `scalar_quantizer_example.mdl`, is the current model. Type `gcs` in the MATLAB Command Window to display the name of your current model.

15 In the **SQ Design Tool** GUI, click the **Design and Plot** button to apply the changes you made to the parameters.

The **SQ Design Tool** GUI should look similar to the following figure.


SQ Design Tool - [Untitled.sqd*]
File Edit Insert Tools Window Help
Performance curve (mean square error at each iteration)
x 10^-5
1.6
1.4
1.2
1
0.8
0.6
Mean Square Error
1 2 3 4 5 6 7 8 9 10
Number of Iterations
Staircase character of the quantizer
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
Final Codewords
-0.8 -0.6 -0.4 -0.2 0 0.2 0.4 0.6 0.8
Final Boundary Points (theoretical bounds are -inf & +inf)
Design and Plot
Export Outputs
Realize Block

Training Set: K

Scalar quantizer

Source of initial codebook: Auto-generate

Number of levels: 128

Initial codebook: [-1.0 : 0.15 : 1.1]

Source of initial boundary points: Mid-points

Initial boundary points (unbounded): [-0.9 : 0.15 : 1.1]

Stopping criteria

Stopping criteria: Relative threshold

Relative threshold: 1e-7

Maximum iteration: 1000

Algorithmic details

Searching methods: Binary search

Tie-breaking rules: Lower indexed codeword

Model

Destination: Current model

Block name: Scalar Quantizer - F

□ Overwrite target block

Total number of iterations = 10

Ready

16 Click the **Realize Block** button.

A new block called Scalar Quantizer - Reflection Coefficients appears in your model file.

17 Click on the **SQ Design Tool** GUI and repeat steps 11-16 for the variable E that represents the residual signal you want to quantize.

A new block called Scalar Quantizer - Residual appears in your model file.

18 Close the **SQ Design Tool** GUI. You do not need to save the SQ Design Tool session.

You have now created two scalar quantizers and added them to your model. In the following topic, "Quantizing an Input Signal" on page 2-57, you learn how to use these scalar quantizers to quantize the residual, the output of the Time-Varying Analysis Filter, and the reflection coefficients, the output of the Levinson-Durbin block.

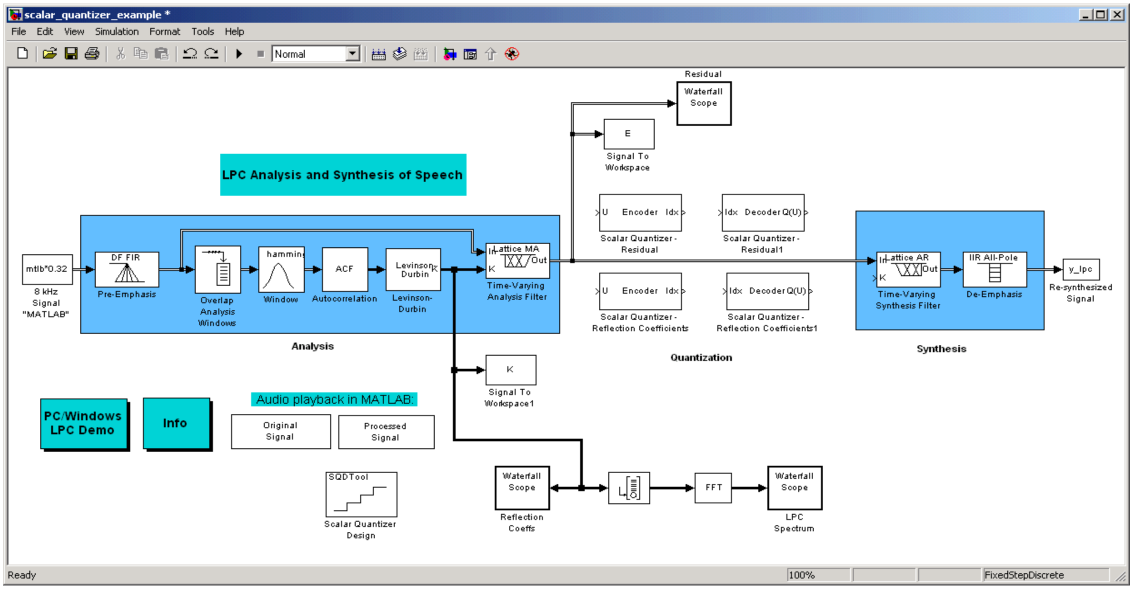
Quantizing an Input Signal

In the previous topic, "Creating a Scalar Quantizer" on page 2-54, you learned how to create scalar quantizers using the Scalar Quantizer Design block. In this topic, you learn how to use these scalar quantizers to quantize the residual and the reflection coefficients in the `dsplpc` demo model. This topic assumes that you completed the procedures described in "Creating a Scalar Quantizer" on page 2-54:

1 Copy the Scalar Quantizer - Reflection Coefficients block, and paste the copy of the block into your model file.

2 Copy the Scalar Quantizer - Residual block, and paste the copy of the block into your model file.

Your model should now look similar to the following.



3 Double-click the Scalar Quantizer - Residual1 block.

4 From the **Quantizer mode** list, choose `Decoder`.

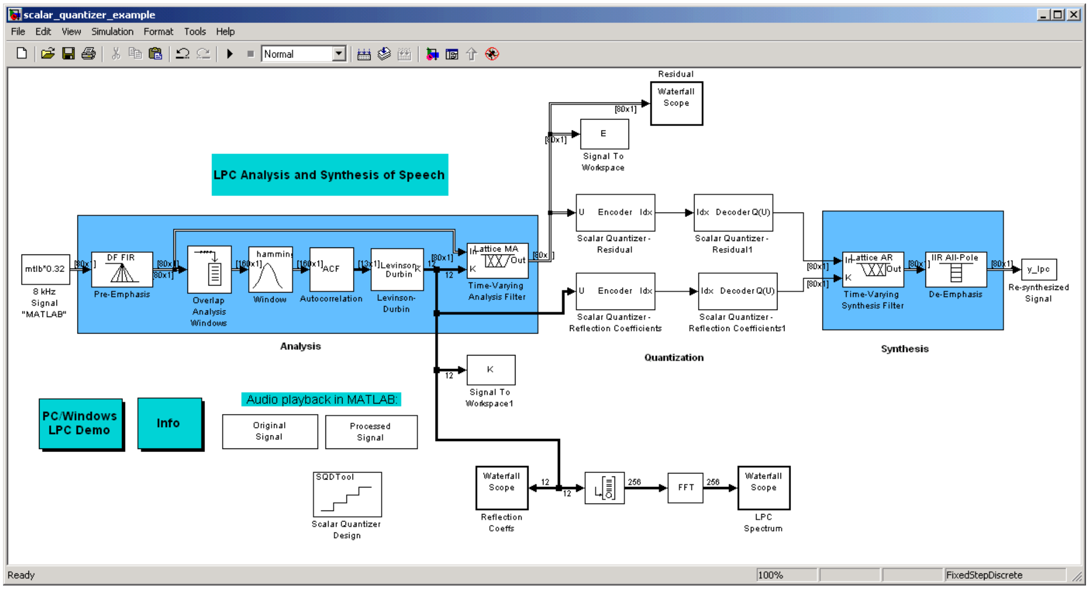
Note that the Scalar Quantizer Design block sets the **Codebook** parameter for you. The Scalar Quantizer Design block has also set the **Boundary points** parameter in the Encoder blocks.

5 Click **OK**.

6 Double-click the Scalar Quantizer - Reflection Coefficients1 block.

7 From the **Quantizer mode** list, choose `Decoder`. Click **OK**.

8 Connect the blocks so your model looks similar to the following figure.



9 Simulate your model.

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

Again, there is almost no difference between the two. You can therefore conclude that quantizing your residual and reflection coefficients did not impact the ability of your system to accurately reproduce the input signal.

In this implementation, the bit rate is 64.4 kbits per second. This is higher than most modern speech coders, which typically have a bit rate of 8 to 64 kbits per second. If you decrease the number of bits allocated for the storage of the reflection coefficients or the residual signal, the speech quality would degrade.

You have now quantized the residual and reflection coefficients in the LPC Analysis and Synthesis of Speech demo model. For more information about quantizers, see the following block reference pages: “Scalar Quantizer” on page 7-653 and “Scalar Quantizer Design” on page 7-661.

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 2-10 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 2-61
- “Deconstructing Multichannel Frame-Based Signals” on page 2-64

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

- “Constructing Multichannel Sample-Based Signals” on page 2-41
- “Constructing Multichannel Frame-Based Signals” on page 2-44

Deconstructing Multichannel Sample-Based Signals

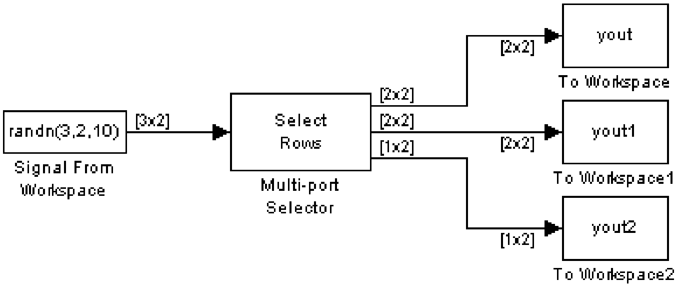
A sample-based signal with M∗N channels is represented by a sequence of M-by-N matrices. (The special case of M = N = 1 represents a single-channel signal.) You can access individual channels of the multichannel signal by using the blocks in the Indexing library (in Signal Management). The following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

- “Deconstructing a Sample-Based Multichannel Signal into Multiple Independent Signals” on page 2-61
- “Deconstructing a Sample-Based Multichannel Signal into a Related Multichannel Signal” on page 2-63

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 2-10 for more about sample-based signals.

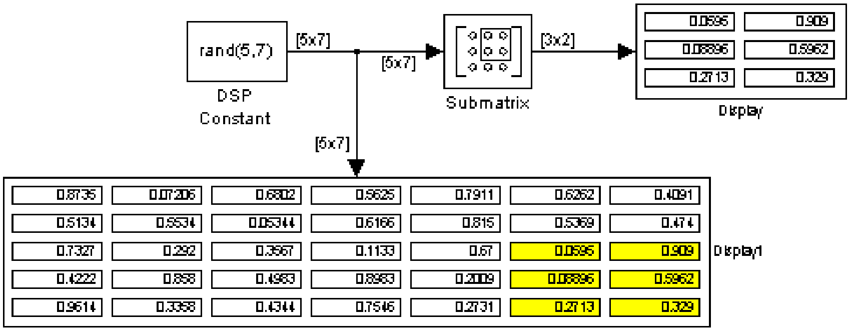
Deconstructing a Sample-Based Multichannel Signal into a Related Multichannel Signal

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.



```
0.0595   0.909
0.08896  0.5962
0.2713   0.329
```

```
0.8735  0.07206  0.6802   0.5625  0.7911  0.6262   0.4091
0.5134  0.5534   0.05344  0.6166  0.815   0.5369   0.474
0.7327  0.292    0.3567   0.1133  0.67    0.0595   0.909
0.4222  0.858    0.4983   0.8983  0.2009  0.08896  0.5962
0.9614  0.3358   0.4344   0.7546  0.2731  0.2713   0.329
```

To build the model, make the following parameter settings:

- In DSP Constant, set **Constant value** = `rand(5,7)`

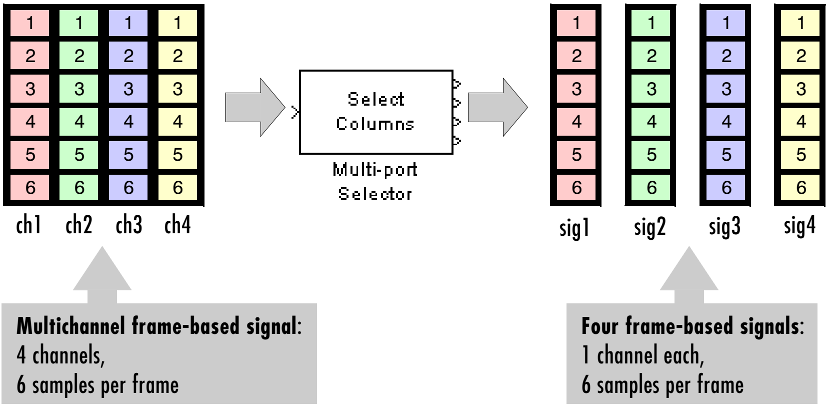
- In Submatrix, set:
  - **Row span** = `Range of rows`
  - **Starting row** = `Index`
  - **Starting row index** = `3`
  - **Ending row** = `Last`
  - **Column span** = `Range of columns`
  - **Starting column** = `Offset from last`
  - **Starting column index** = `1`
  - **Ending column** = `Last`

See “Sample-Based Multichannel Signals” on page 2-10 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:

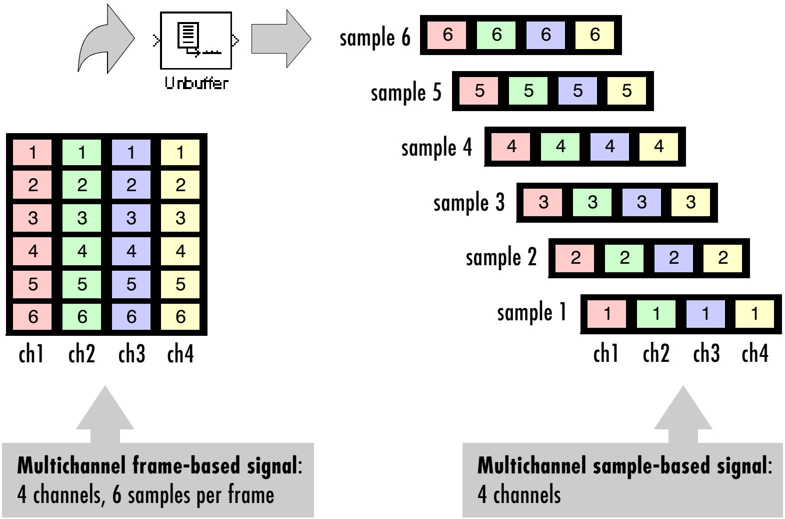
- *Split the channels into independent signals* — The constituent channels of a multichannel frame-based signal can be extracted to form individual frame based signals (with the same frame rate and size) by using the Multiport Selector block in the Indexing library (in Signal Management).



Multichannel frame-based signal:
4 channels,
6 samples per frame

Four frame-based signals:
1 channel each,
6 samples per frame

- *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 2-65
- “Unbuffering a Frame-Based Signal into a Sample-Based Signal” on page 2-67

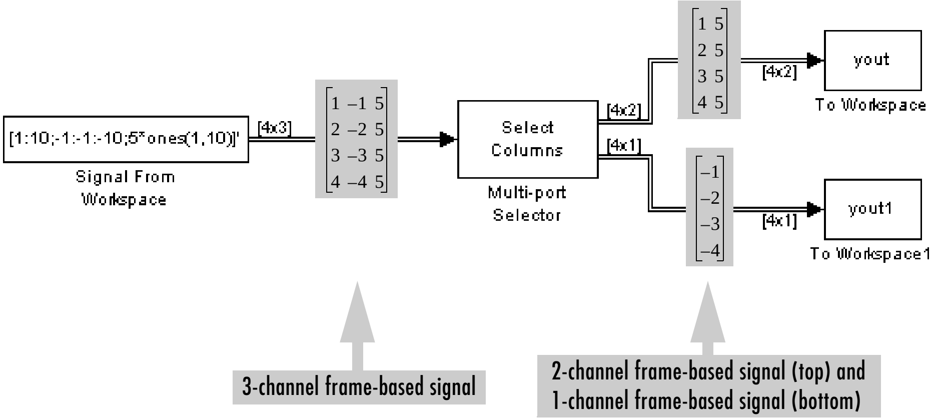
The final section explains how to reorder the channels in a frame-based signal *without* splitting the channels apart:

- “Reordering Channels in a Frame-Based Multichannel Signal” on page 2-68

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 2-11 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 these sections for more information:

- “Creating Signals Using the Signal From Workspace Block” on page 2-37
- “Importing a Multichannel Frame-Based Signal” on page 2-75

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).

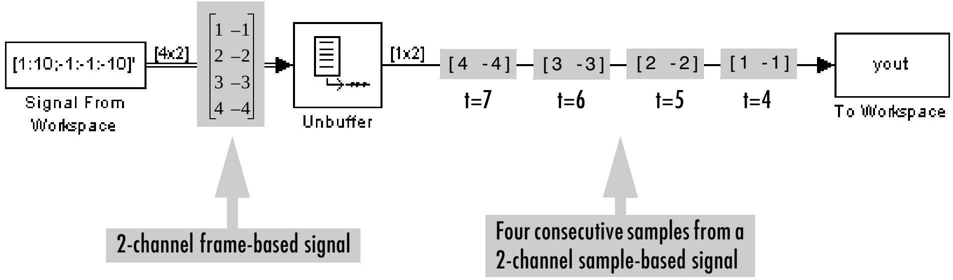
The Unbuffer block performs the inverse operation of the Buffer block's "sample-based to frame-based" buffering process, and generates an N-channel sample-based output from an N-channel frame-based input. The first row in each input matrix is always the first sample-based output. In other words, the 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 2-19 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:

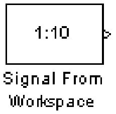
- “Importing a Multichannel Sample-Based Signal” on page 2-69
- “Importing a Multichannel Frame-Based Signal” on page 2-75
- “Importing WAV Files” on page 2-78

For information about creating signals, see these sections:

- “Creating Signals Using Constant Blocks” on page 2-32
- “Creating Signals Using Signal Generator Blocks” on page 2-35
- “Creating Signals Using the Signal From Workspace Block” on page 2-37

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.



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 2-75 for more information. The

Sample-time parameter specifies the sample period of the sample-based output. See "Sample-Based Multichannel Signals" on page 2-10 for general information about sample-based signals.

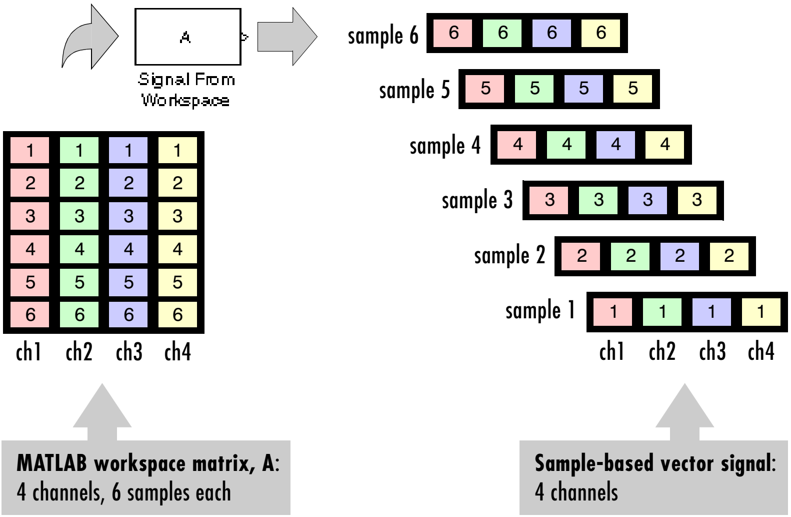
The following sections explain how the Signal From Workspace generates its output:

- "Importing a Sample-Based Vector Signal" on page 2-70
- "Importing a Sample-Based Matrix Signal" on page 2-72

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.



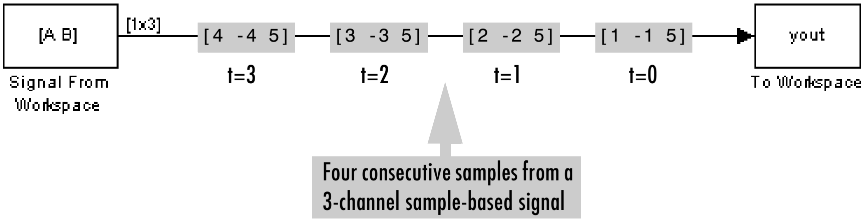
As the figure above suggests, the output of the Signal From Workspace block can only be a valid sample-based signal (having N independent channels) if the M-by-N workspace matrix A in fact represents N independent channels, each containing M consecutive samples. In other words, the workspace matrix must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

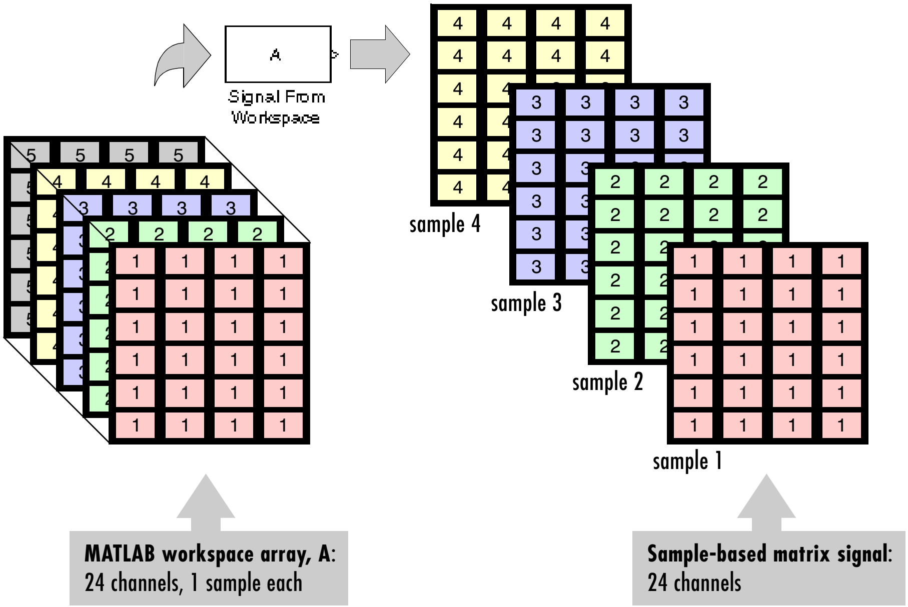
```
C = [A B]
```

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 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`.


A
Signal From Workspace
sample 4
sample 3
sample 2
sample 1
MATLAB workspace array, A:
24 channels, 1 sample each
Sample-based matrix signal:
24 channels

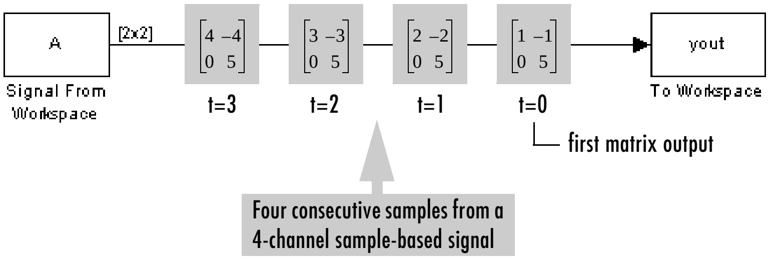
As the figure above suggests, the output of the Signal From Workspace block can only be a valid sample-based signal (having M∗N independent channels) if the M-by-N-by-P workspace array A in fact represents M∗N independent channels, each having P samples. In other words, the workspace array must be oriented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


A
[2x2]
Signal From Workspace
$\begin{bmatrix} 4 & -4 \\ 0 & 5 \end{bmatrix}$
t=3
$\begin{bmatrix} 3 & -3 \\ 0 & 5 \end{bmatrix}$
t=2
$\begin{bmatrix} 2 & -2 \\ 0 & 5 \end{bmatrix}$
t=1
$\begin{bmatrix} 1 & -1 \\ 0 & 5 \end{bmatrix}$
t=0
yout
To Workspace

first matrix output

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

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**

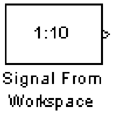
The **Sample time** and **Samples per frame** settings of 1 yield a sample-based output with sample period of 1 second. Each of the four elements in the matrix represents an independent channel. The **Form output after final data value** parameter setting specifies that all outputs after the one-hundredth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

These two sections may also be of interest to you:

- "Creating Signals Using the Signal From Workspace Block" on page 2-37
- "Constructing Multichannel Sample-Based Signals" on page 2-41

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 2-69 for more information.

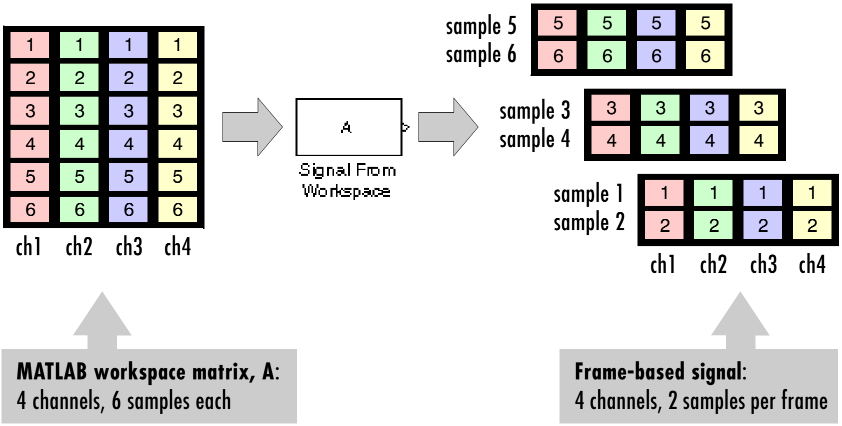
The **Sample-time** parameter specifies the sample period, T_s, of the frame-based output. The frame period of the signal is $M*T_s$, where M is the value of the **Samples per frame** parameter. See "Frame-Based Multichannel Signals" on page 2-11 for general information about frame-based signals.

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 (that is, 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.

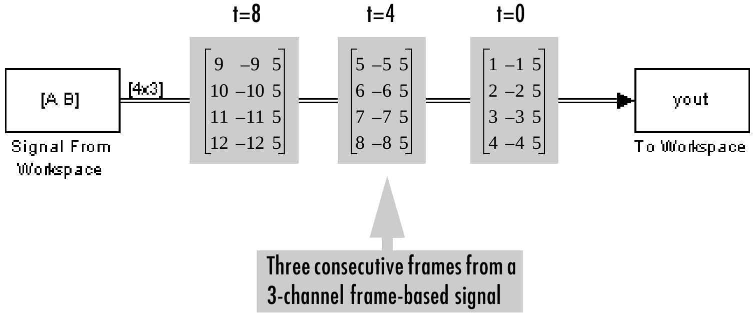
When the block has output all of the rows available in the specified variable, it can start again at the beginning of the signal, or simply repeat the final value (or generate zeros) until the end of the simulation. This behavior is controlled by the **Form output after final data value by** parameter.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



Three consecutive frames from a 3-channel frame-based signal

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 2-79
- "Exporting and Playing WAV Files" on page 2-86

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.

yout
To 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 2-19 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 2-80
- “Exporting a Frame-Based Signal Using the To Workspace Block” on page 2-83

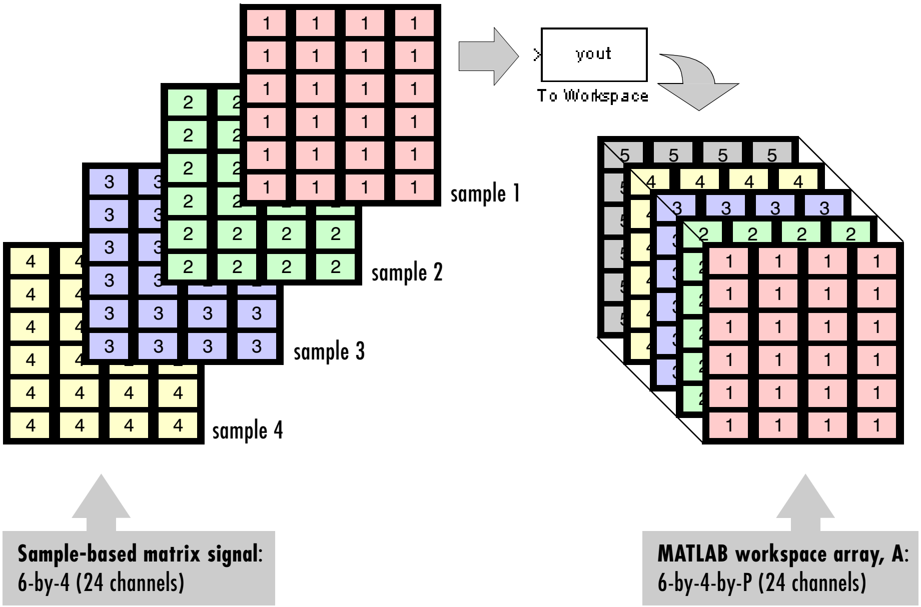
These sections may also be of interest:

- “Creating Signals Using the Signal From Workspace Block” on page 2-37
- “Constructing Multichannel Sample-Based Signals” on page 2-41

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.


yout
To Workspace
sample 1
sample 2
sample 3
sample 4
Sample-based matrix signal:
6-by-4 (24 channels)
MATLAB workspace array, A:
6-by-4-by-P (24 channels)

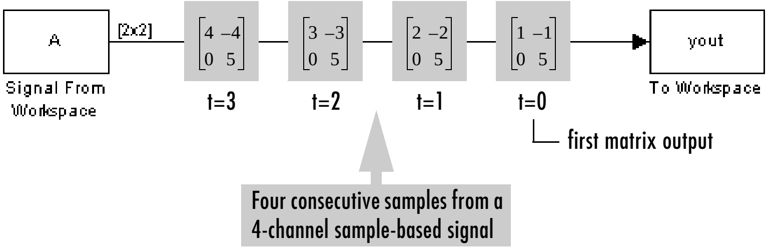
The workspace array always has time running along its third (P) dimension. Samples are saved along the P dimension whether the input is a matrix, vector, or scalar (single channel).

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

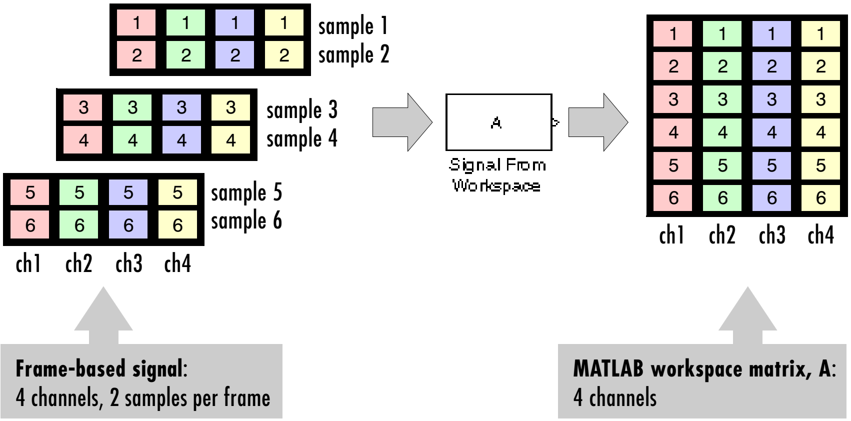
Run the model, and look at output yout. The first four samples (pages) are shown below:

```
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.



Frame-based signal: 4 channels, 2 samples per frame

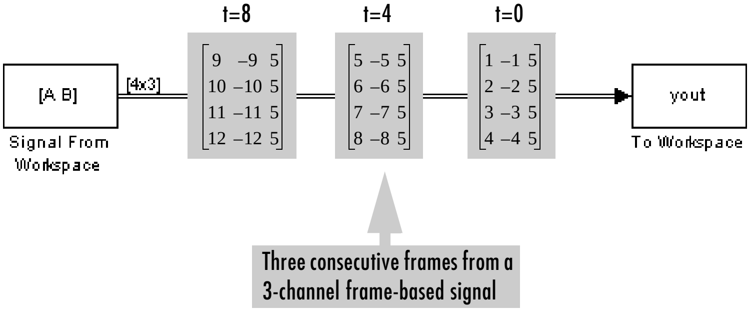
MATLAB workspace matrix, A: 4 channels

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,...



Three consecutive frames from a 3-channel frame-based signal

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 2-37
- “Constructing Multichannel Sample-Based Signals” on page 2-41

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.

These demos may also be of interest:

- Audio Flanger — PC/Windows
- Demonstration of Audio Reverberation
- LPC Analysis and Synthesis of Speech — 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:

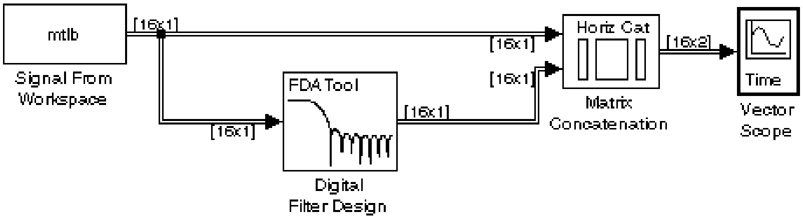
- "Displaying Signals in the Time-Domain" on page 2-87
- "Displaying Signals in the Frequency-Domain" on page 2-89
- "Displaying Matrices" on page 2-90

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.

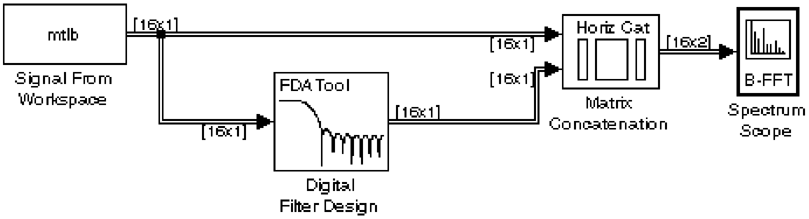
- Select **Compact display** from the right-click menu to allow the scope to use all the available space in the window.
- Select **CH 1** from the right-click menu, and then select **Marker** and "**o**" from the submenus, to mark the data points on the channel 1 signal with circles.
- Select **CH 1**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

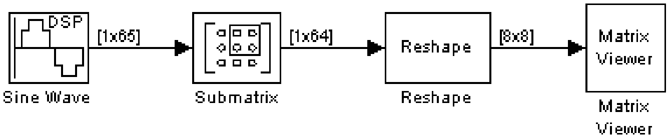
You may want to set the **Stop time** in the **Simulation Parameters** dialog box to `inf` to allow the simulation to run longer. See “Improving the Appearance of the Display” on page 2-88 for some tips on improving the scope display.

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 2-92
- "Algorithmic Delay" on page 2-93

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 2-13 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 7, "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)

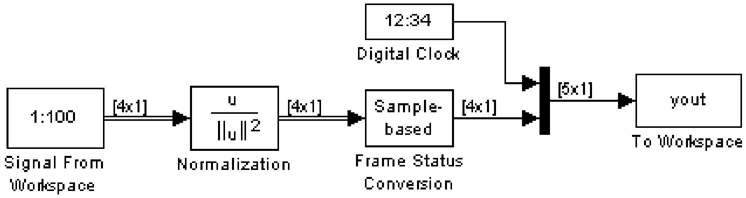
The following sections explain the different levels of delay, and how the simulation and parameter settings can affect the level of delay that a particular block experiences.

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 7, "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,

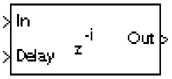
```
[1 2 3 4]' ./ sum([1 2 3 4]'.^2)
```

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.

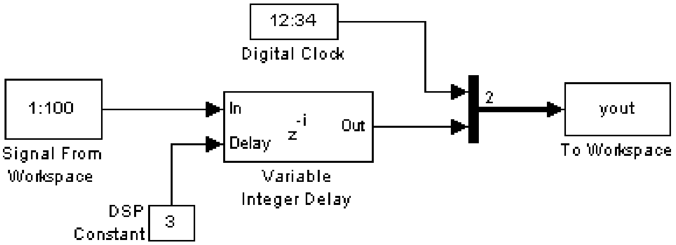
Basic Algorithmic Delay

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.



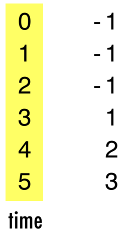
Use the default settings for the Digital Clock, Mux, and To Workspace blocks, and adjust the Signal From Workspace block's parameters to the values below:

- **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
```

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.

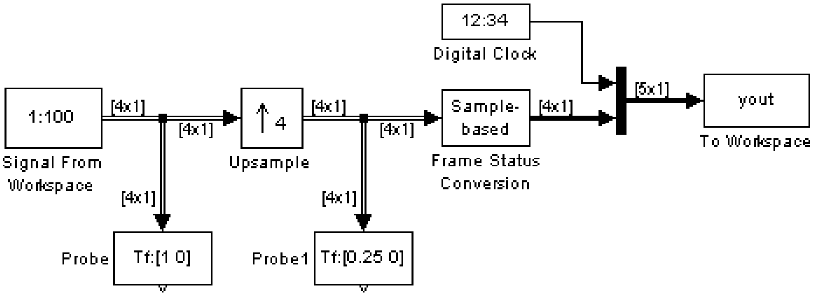
Block Sample Mode. Many blocks can operate in either sample-based or frame-based modes. In source blocks, the mode is usually determined by the **Samples per frame** parameter; a value of 1 for this parameter indicates sample-based mode, while a value greater than 1 indicates frame-based mode. In nonsource 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 2-100
- "Example: Zero Tasking Latency" on page 2-102

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 (that is, 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

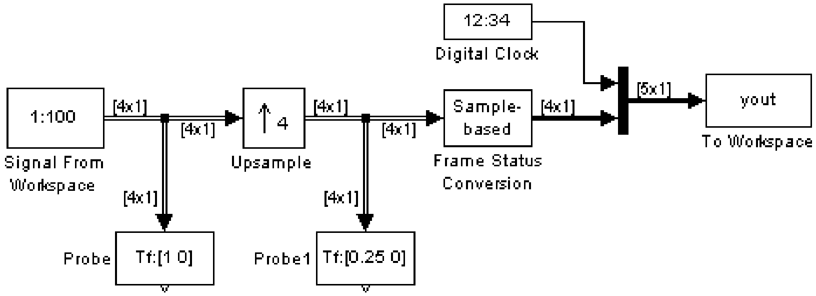
```
squeeze(yout)'
```

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.

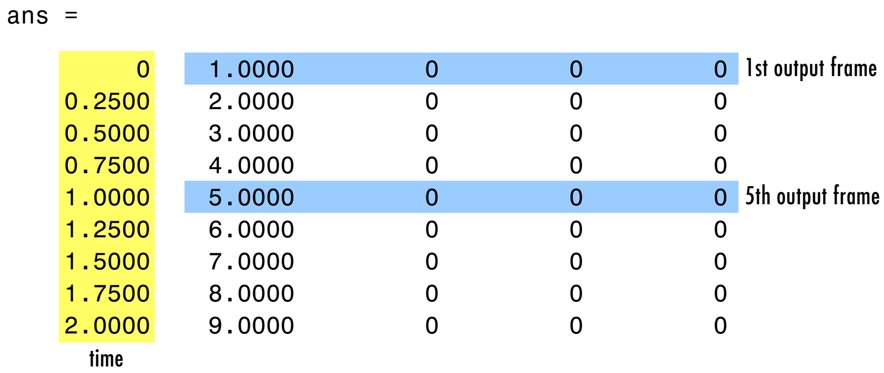


To engage the Simulink single-tasking mode, adjust the following settings in the **Solver** panel of the **Simulation Parameters** dialog box:

- **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
   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
   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
```

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).

3

Filters

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

Digital Filter Block

You can use the Digital Filter block to implement digital FIR and IIR filters in your models. Use this block if you have already performed the design and analysis and are able to provide the filter coefficients directly. You can use this block to filter single-channel and multichannel signals, and to simulate floating-point and fixed-point filters. You can use Real-Time Workshop to generate highly optimized C code from your filter block. For more information on generating C code from models, see Appendix B, “Code Generation Support.”

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

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

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

This section includes the following topics:

- “Implementing a Lowpass Filter” on page 3-2 — Create a lowpass filter using the Digital Filter block
- “Implementing a Highpass Filter” on page 3-5 — Create a highpass filter using the Digital Filter block
- “Filtering High-Frequency Noise” on page 3-7 — Build a system capable of filtering high-frequency noise using a highpass and lowpass filter

Implementing a Lowpass Filter

The Digital Filter block, in the DSP Blockset Filter Designs library, is useful for implementing a digital FIR or IIR filter. In this example, you create a lowpass filter using the Digital Filter block.

To learn how to create a highpass filter, see “Implementing a Highpass Filter” on page 3-5. To design and implement a new filter, see “Digital Filter Design Block” on page 3-13:

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

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

You could have calculated these values using

- 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});
```

- Digital Filter Design block — Design a lowpass filter as in “Creating a Lowpass Filter” on page 3-16. Then export the filter coefficients to the MATLAB workspace to the variable `lopassNum` as described in the “Importing and Exporting Quantized Filters” section of the Filter Design Toolbox documentation.

2 Open Simulink and create a new model file.

3 In the Simulink Model browser, double-click **DSP Blockset**.

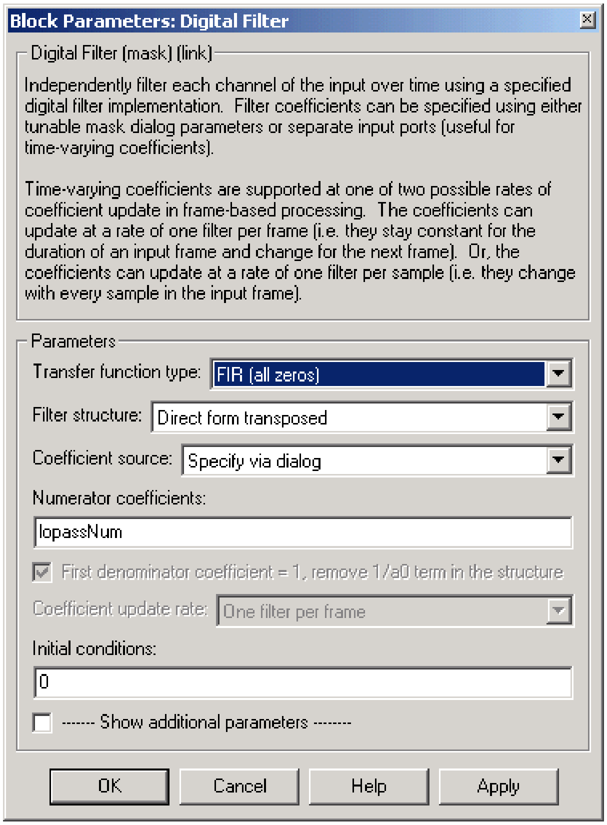
4 Double-click **Filtering**, and select the **Filter Designs** library.

5 Click-and-drag a Digital Filter block into your model.

6 Double-click the Digital Filter block.

The **Block Parameters: Digital Filter** dialog box opens.

7 Set the parameters to the values shown below.



Block Parameters: Digital Filter

Digital Filter (mask) (link)

Independently filter each channel of the input over time using a specified digital filter implementation. Filter coefficients can be specified using either tunable mask dialog parameters or separate input ports (useful for time-varying coefficients).

Time-varying coefficients are supported at one of two possible rates of coefficient update in frame-based processing. The coefficients can update at a rate of one filter per frame (i.e. they stay constant for the duration of an input frame and change for the next frame). Or, the coefficients can update at a rate of one filter per sample (i.e. they change with every sample in the input frame).

Parameters

Transfer function type: FIR (all zeros)

Filter structure: Direct form transposed

Coefficient source: Specify via dialog

Numerator coefficients:

lopassNum

☑ First denominator coefficient = 1, remove 1/a0 term in the structure

Coefficient update rate: One filter per frame

Initial conditions:

0

□ ------- Show additional parameters --------

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

- Type in a variable name from the MATLAB workspace, such as `lopassNum`.
- Type in filter design commands from the Signal Processing Toolbox or the Filter Design Toolbox, such as `fir1(5, 0.2, 'low')`.
- Type in a vector of the filter coefficient values.

8 Click **OK**.

The Digital Filter block in your model now represents a lowpass filter.

Implementing a Highpass Filter

In the previous topic, "Implementing a Lowpass Filter" on page 3-2, you learned how to create a lowpass filter. Now, create a highpass filter using the Digital Filter block:

1 Define the highpass filter coefficients in the MATLAB workspace by typing

```
highpassNum = [-0.0051 0.0181 -0.0069 -0.0283 -0.0061 0.0549
0.0579   -0.0826 -0.2992 0.5946 -0.2992 -0.0826 0.0579 0.0549
-0.0061 -0.0283   -0.0069 0.0181 -0.0051];
```

You could have calculated these values using

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

```
[N Fo Ao W] = remezord([0.2 0.5],[0 1], ...
[1.000000000000001e-004 5.750112778453722e-002]);
hipassNum = remez(N,Fo,Ao,W,{16});
```

- Digital Filter Design block — Design a highpass filter as in "Creating a Highpass Filter" on page 3-18. Then export the filter coefficients to the MATLAB workspace to the variable `highpassNum` as described in the "Importing and Exporting Quantized Filters" section of the Filter Design Toolbox documentation.

2 Open Simulink and create a new model file.

If you completed the procedure in "Implementing a Lowpass Filter" on page 3-2, you can continue to use your the same model file.

3 In the Simulink Model browser, double-click **DSP Blockset**.

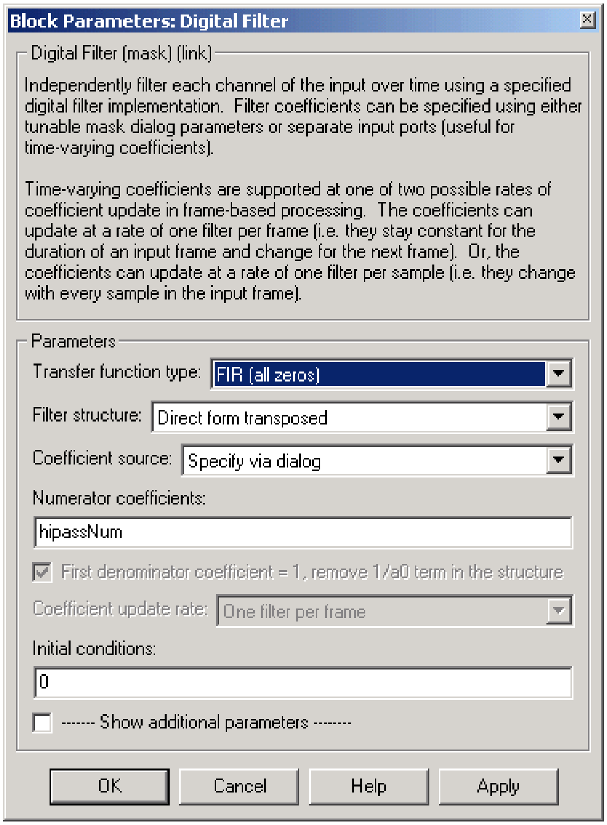
4 Double-click **Filtering**, and select the **Filter Designs** library.

5 Click-and-drag a Digital Filter block into your model.

6 Double-click the Digital Filter block.

The **Block Parameters: Digital Filter** dialog box opens.

7 Set the parameters to the values shown below.



Block Parameters: Digital Filter

Digital Filter (mask) (link)

Independently filter each channel of the input over time using a specified digital filter implementation. Filter coefficients can be specified using either tunable mask dialog parameters or separate input ports (useful for time-varying coefficients).

Time-varying coefficients are supported at one of two possible rates of coefficient update in frame-based processing. The coefficients can update at a rate of one filter per frame (i.e. they stay constant for the duration of an input frame and change for the next frame). Or, the coefficients can update at a rate of one filter per sample (i.e. they change with every sample in the input frame).

Parameters

Transfer function type: FIR (all zeros)

Filter structure: Direct form transposed

Coefficient source: Specify via dialog

Numerator coefficients:

hipassNum

☑ First denominator coefficient = 1, remove 1/a0 term in the structure

Coefficient update rate: One filter per frame

Initial conditions:

0

□ ------- Show additional parameters -------

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

- Type in a variable name from the MATLAB workspace, such as `highpassNum`.
- 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.

8 Click **OK**.

The Digital Filter block in your model now represents a highpass filter.

Filtering High-Frequency Noise

In the model, `digitalfilter_tut.mdl`, high-frequency noise is output by a highpass filter excited by a uniform random signal. This high frequency noise is added to a sine wave. This noisy sine wave is fed through a lowpass filter that filters out the high-frequency noise. In this procedure, you create this model and simulate its behavior.

Note You can open many of the example models in the online version of this document by clicking 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.

---

1 Open the model that contains the highpass and lowpass filter described in “Implementing a Lowpass Filter” on page 3-2 and “Implementing a Highpass Filter” on page 3-5.

2 Click and drag the blocks in this table into your model file.

Block	Library	Quantity
Matrix Concatenation	Math Functions / Matrices and Linear Algebra / Matrix Operations	1
Random Source	DSP Sources	1
Sine Wave	DSP Sources	1
Sum	The Simulink Math Operations library	1
Vector Scope	DSP Sinks	1

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

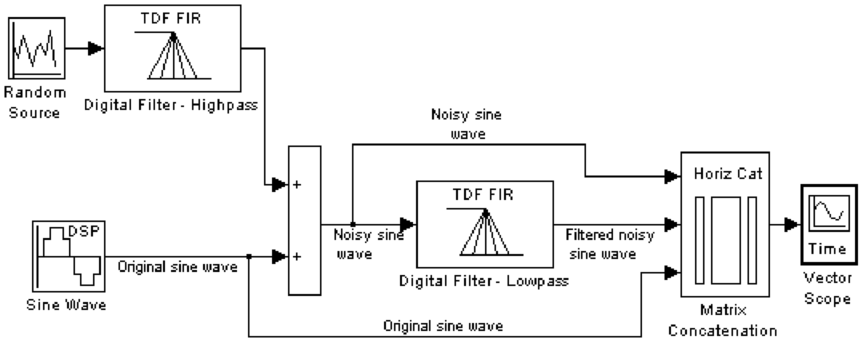
Parameter Settings for the Other Blocks

Block	Parameter Setting
Matrix Concatenation	• **Number of inputs** — `3` • **Concatenation method** — `Horizontal`
Random Source	• **Source type** — `Uniform` • **Minimum** — `0` • **Maximum** — `4` • **Sample mode** — `Discrete` • **Sample time** — `1/1000` • **Samples per frame** — `50`
Sine Wave	• **Frequency (Hz)** — `75` • **Sample time** — `1/1000` • **Samples per frame** — `50`

Parameter Settings for the Other Blocks (Continued)

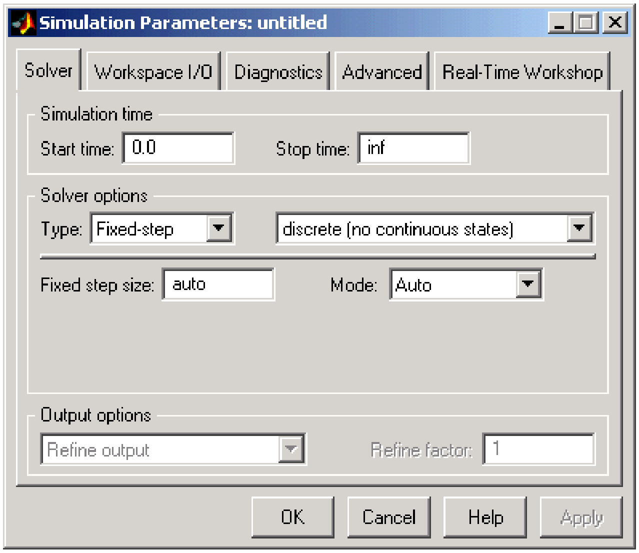
Block	Parameter Setting
Sum	• **Icon shape** — `rectangular` • **List of signs** — `++`
Vector Scope	**Scope properties:** • **Input domain** — `Time` • **Time display span (number of frames)** — `1`

4 Connect the blocks as shown in the following figure. You might need to resize some of the blocks to make your model look like the figure.



5 From the Simulation menu, choose **Simulation parameters**.

6 Set the Simulation parameters as indicated in the following figure.



Simulation time

Start time: 0.0

Stop time: inf

Solver options

Type: Fixed-step

discrete (no continuous states)

Fixed step size: auto

Mode: Auto

Output options

Refine output

Refine factor: 1

7 Click **OK**.

8 In the model window, from the **Simulation** menu, choose **Start**.

The model simulation begins and the Scope displays the three input signals. When you finish observing the running model, from the **Simulation** menu, choose **Stop**.

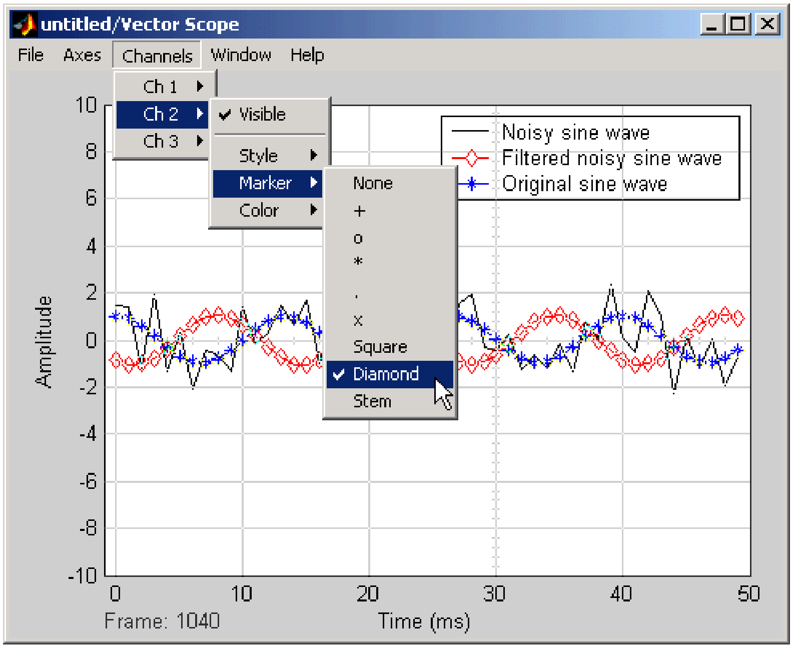
9 Double-click the Vector Scope block and select the **Show display properties** check box. Select the **Channel legend** check box and click **OK**.Next time you run the simulation, a legend appears in the Vector Scope window.

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

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

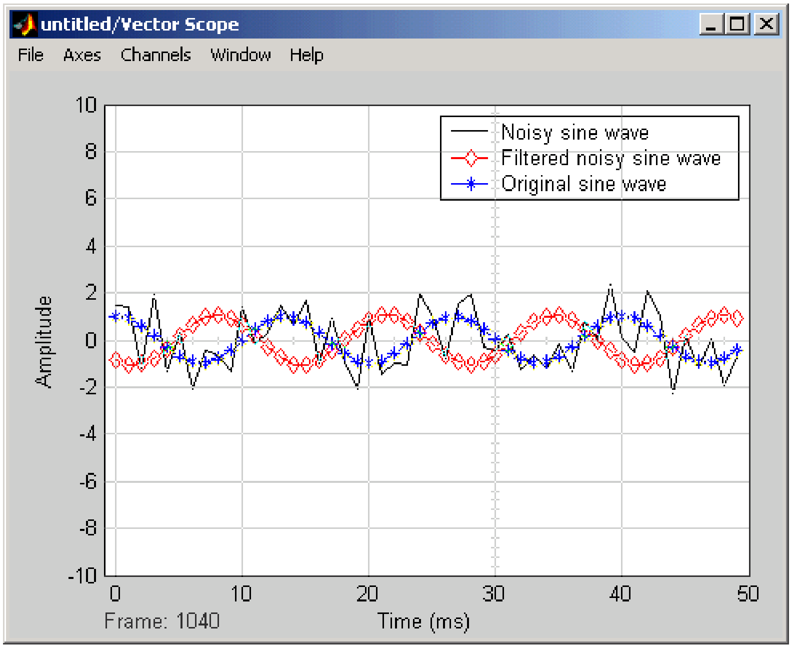
Point to **Ch 2** and set the **Style** to **-**, **Marker** to **Diamond**, and **Color** to **Red**.

Point to **Ch 3** and set the **Style** to **None**, **Marker** to *, and **Color** to **Blue**.



11 Rerun the simulation and compare the original sine wave, noisy sine wave, and filtered noisy sine wave in the **Vector Scope** display.

You can see that the lowpass filter filters out the high-frequency noise in the noisy sine wave.



Digital Filter Design Block

You can use the Digital Filter Design block to design and implement a digital filter. The filter you design can filter single-channel or multichannel signals. The Digital Filter Design block is ideal for simulating the numerical behavior of your filter on a single- or double-precision floating-point system, such as a personal computer or DSP chip. You can use Real-Time Workshop to generate C code from your filter block. For more information on generating C code from models, see Appendix B, “Code Generation Support.”

This section includes the following topics:

- “Overview of the Digital Filter Design Block” on page 3-13 — Learn the basic functionality of the Digital Filter Design block
- “Choosing Between Filter Design Blocks” on page 3-14 — Determine whether the Digital Filter Design block or the Filter Realization Wizard is right for your application
- “Creating a Lowpass Filter” on page 3-16 — Use the Digital Filter Design block to design and implement a lowpass filter
- “Creating a Highpass Filter” on page 3-18 — Use the Digital Filter Design block to design and implement a highpass filter
- “Filtering High-Frequency Noise” on page 3-20 — Create a system capable of filtering high-frequency noise using a highpass and a lowpass filter

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

Overview of the Digital Filter Design Block

Filter Design and Analysis. You perform all filter design and analysis within the Filter Design and Analysis Tool (FDATool) GUI, which opens when you double-click the Digital Filter Design block. FDATool provides extensive filter design parameters and analysis tools such as pole-zero and impulse response plots.

Filter Implementation. Once you have designed your filter using FDATool, the block automatically realizes the filter using the filter structure you specified. You can then use the block to filter signals in your model. You can also fine-tune the filter by changing the filter specification parameters during a simulation. The outputs of the Digital Filter Design block numerically match the outputs of the `filter` function in the Filter Design Toolbox and the `filter` function in 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.

To learn how to save your filter designs, see the “Saving and Opening Filter Design Sessions” section of the Signal Processing Toolbox documentation. To learn how to import and export your filter designs, see the “Importing and Exporting Quantized Filters” section of the Filter Design Toolbox documentation.

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

See the block reference page, “Digital Filter Design” on page 7-212, for more information. For information on choosing between the Digital Filter Design block and the Filter Realization Wizard, see “Choosing Between Filter Design Blocks” on page 3-14.

Choosing Between Filter Design Blocks

The Digital Filter Design block and Filter Realization Wizard block both let you design and implement digital filters. This topic explains the similarities and differences between the Digital Filter Design block and the Filter Realization Wizard. In addition, this topic covers how to choose the block that is best suited for your needs.

Similarities

The Digital Filter Design Block and Filter Realization Wizard are similar in the following ways:

- Filter design and analysis options — Both blocks 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

The Digital Filter Design Block and Filter Realization Wizard handle the following things differently:

- 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.
- 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 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

The following are specific situations where only the Digital Filter Design block or the Filter Realization Wizard is appropriate.

- Digital Filter Design
  - Use to simulate single- and double-precision floating-point filters.
  - Use to filter multichannel signals.
  - Use to generate highly optimized ANSI/ISO C code that implements floating-point filters for embedded systems. For more information on code generation, see 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 (the block builds the filter from Sum, Gain, and Delay blocks).

See “Filter Realization Wizard” on page 3-26 for information about this block.

Creating a Lowpass Filter

The Digital Filter Design block, located in the DSP Blockset Filter Designs library, is useful for designing and implementing a digital FIR or IIR filter. In this example, you create a lowpass filter using the Digital Filter Design block. To implement a filter you already designed, see “Digital Filter Block” on page 3-2:

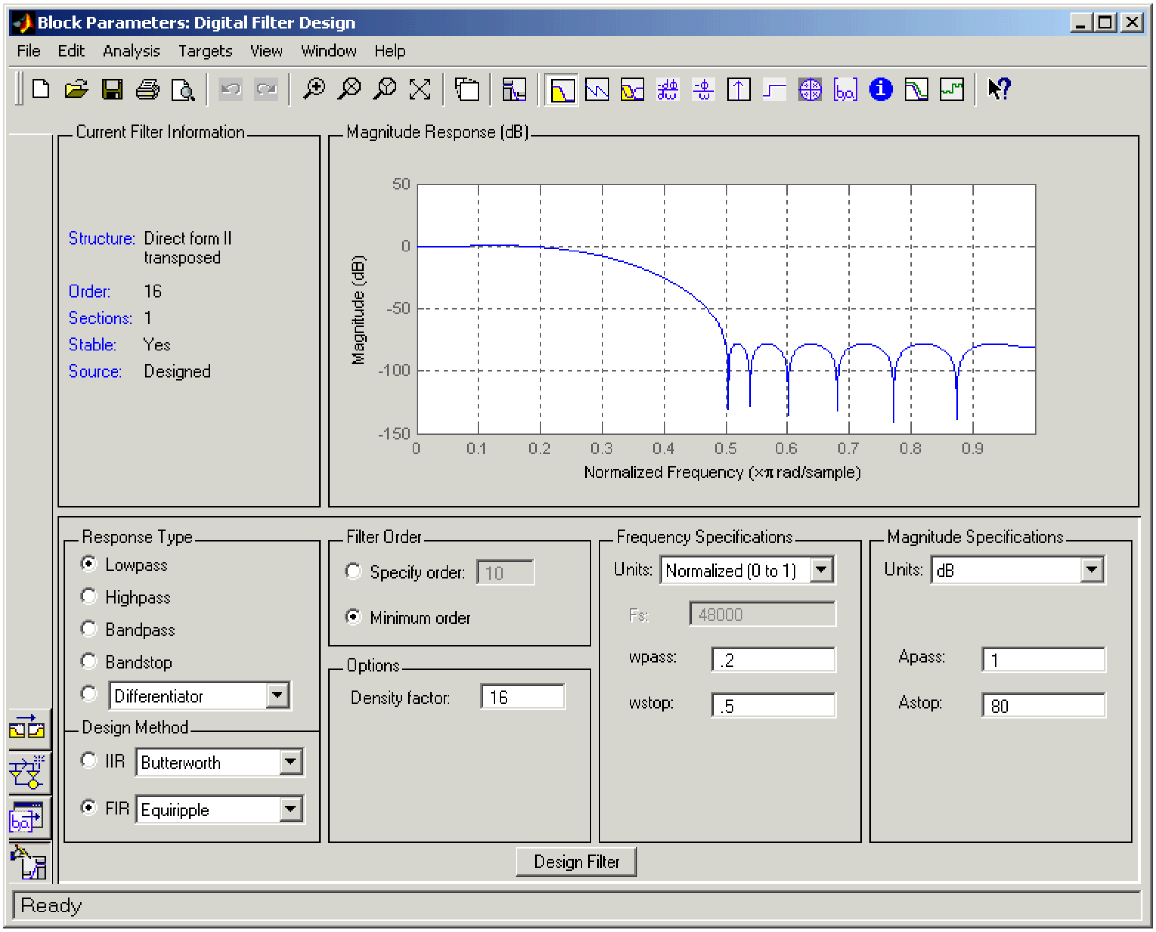
1. To create the lowpass filter in the model, open Simulink and create a new model file.

2. From the Filter Designs library, drag a Digital Filter Design block into your new model

3. Double-click the Digital Filter Design block.

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

4 Set the block parameters of the GUI as shown in the following figure.


Block Parameters: Digital Filter Design
File Edit Analysis Targets View Window Help
Magnitude Response (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
Normalized Frequency (×π rad/sample)
Design Filter

Current Filter Information

Structure: Direct form II transposed
Order: 16
Sections: 1
Stable: Yes
Source: Designed

Response 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: .2
wstop: .5

Magnitude Specifications

Units: dB
Apass: 1
Astop: 80

Ready

5 Click **Design Filter** at the bottom of the GUI to design the filter.

Your Digital Filter Design block now represents a filter with the parameters you specified. You can explore the other buttons, which provide other filter analysis tools such as pole-zero plots and impulse response plots. Click the Magnitude Response button to view the original display.

6 In the **Edit** menu, select **Convert Structure**.

   The **Convert Structure** dialog box opens.

7 Select **Direct-Form FIR Transposed** and click **OK**.

The Digital Filter Design block now represents a lowpass filter with a Direct-Form FIR Transposed 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.

To learn how to create a highpass filter using the Digital Filter Design block, see "Creating a Highpass Filter" on page 3-18.

Creating a Highpass Filter

The Digital Filter Design block, located in the DSP Blockset Filter Designs library, is useful for designing and implementing a digital FIR or IIR filter. In this example, you create a highpass filter using the Digital Filter Design block. To implement a filter you already designed, see "Digital Filter Block" on page 3-2:

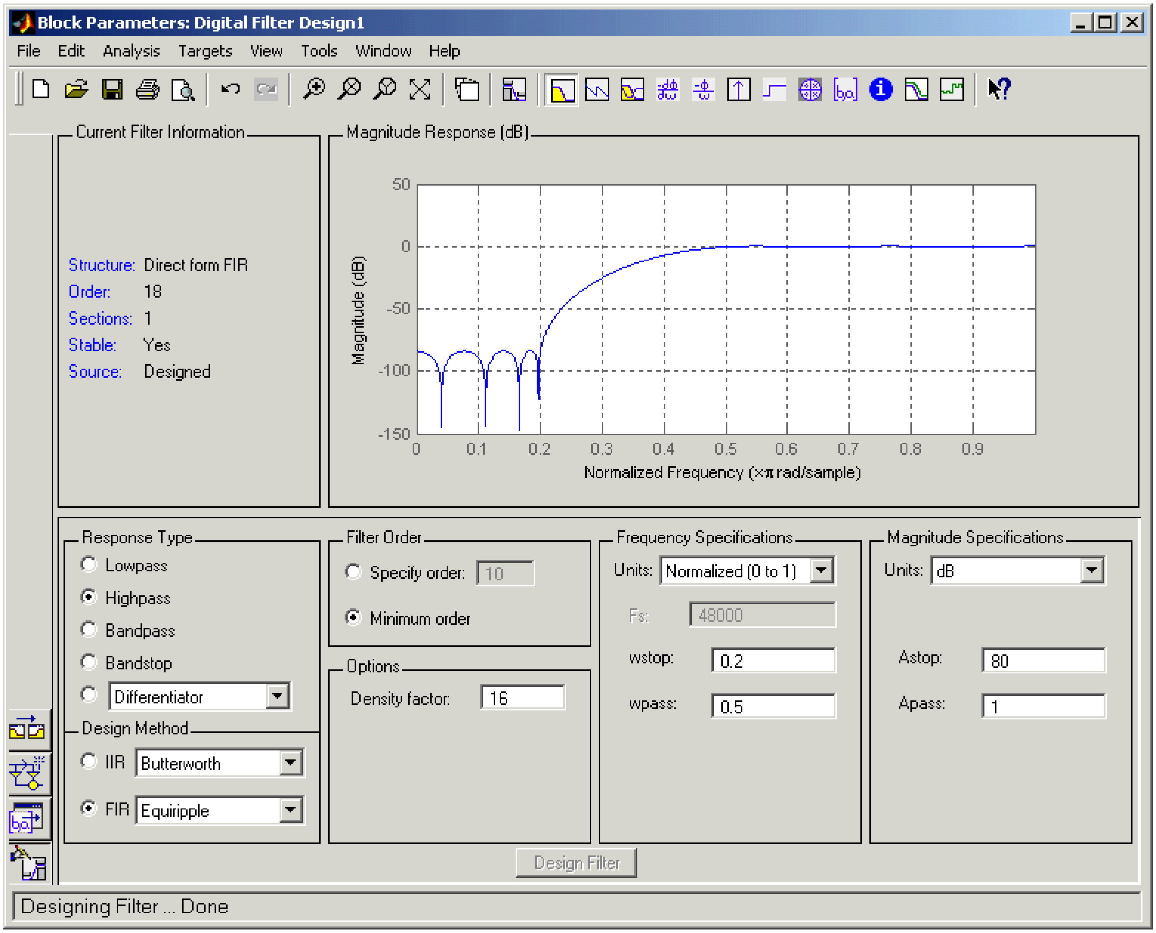
1 From the Filter Designs library, drag a Digital Filter Design block into a Simulink model.

   If you created the model in "Creating a Lowpass Filter" on page 3-16, you can place this block into the same model file.

2 Double-click the new Digital Filter Design block.

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

3 Set the block parameters of the GUI as shown in the following figure.



Block Parameters: Digital Filter Design1
File Edit Analysis Targets View Tools Window Help
Current Filter Information
Magnitude Response (dB)
50
0
-50
-100
-150
Magnitude (dB)
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9
Normalized Frequency (×π rad/sample)
Design Filter

Current Filter Information

Structure: Direct form FIR
Order: 18
Sections: 1
Stable: Yes
Source: Designed

Response 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

Designing Filter ... Done

4 Click the **Design Filter** button at the bottom of the GUI to design the filter.

Your Digital Filter Design block now represents a filter with the parameters you specified.

5 In the **Edit** menu, select **Convert Structure**.

The **Convert Structure** dialog box opens.

6 Select **Direct-Form FIR Transposed** and click **OK**.

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 described in "Creating a Lowpass Filter" on page 3-16. The highpass filter passes the frequencies stopped by the lowpass filter, and stops the frequencies passed by the lowpass filter. In the next topic, you learn how to build a model where the lowpass filter filters out the high-frequency noise output by this highpass filter.

To learn how to create a model capable of filtering high frequency noise, see "Filtering High-Frequency Noise" on page 3-20.

Filtering High-Frequency Noise

In this topic, you create a model that simulates high-frequency noise being output by a highpass filter that is excited by a uniform random signal. This high frequency noise is added to a sine wave. This noisy sine wave is fed through a lowpass filter that filters out the high-frequency noise.

Note You can open many of the example models in the online version of this document by clicking 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.

---

1 Open the model that contains the highpass and lowpass filter described in "Creating a Lowpass Filter" on page 3-16 and "Creating a Highpass Filter" on page 3-18.

2 Click and drag the blocks in this table into your model file.

Block	**Library**	**Quantity**
Matrix Concatenation	Math Functions / Matrices and Linear Algebra / Matrix Operations	1
Random Source	DSP Sources	1
Sine Wave	DSP Sources	1
Sum	The Simulink Math Operations library	1
Vector Scope	DSP Sinks	1

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

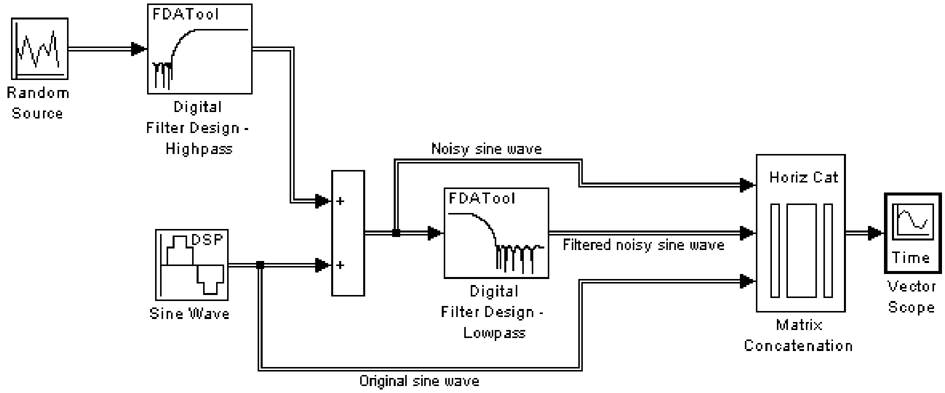
Parameter Settings for the Other Blocks

Block	**Parameter Setting**
Matrix Concatenation	• **Number of inputs** — `3` • **Concatenation method** — `Horizontal`
Random Source	• **Source type** — `Uniform` • **Minimum** — `0` • **Maximum** — `4` • **Sample mode** — `Discrete` • **Sample time** — `1/1000` • **Samples per frame** — `50`
Sine Wave	• **Frequency (Hz)** — `75` • **Sample time** — `1/1000` • **Samples per frame** — `50`

Parameter Settings for the Other Blocks (Continued)

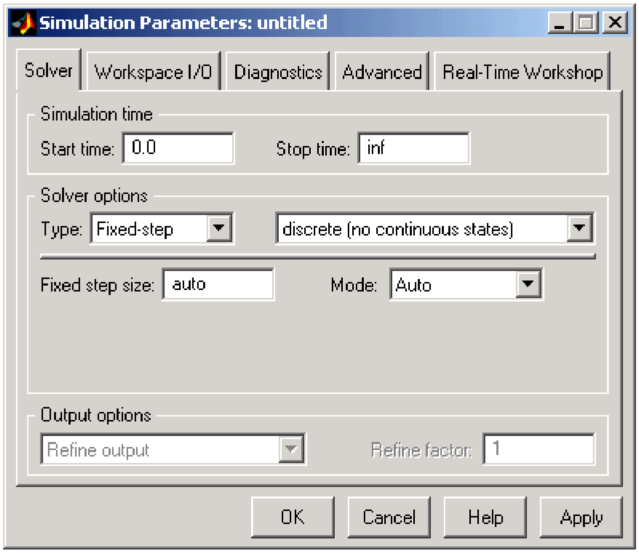
Block	Parameter Setting
Sum	• **Icon shape** — `rectangular` • **List of signs** — `++`
Vector Scope	**Scope properties:** • **Input domain** — `Time` • **Time display span (number of frames)** — `1`

4 Connect the blocks as shown in the following figure. You might need to resize some of the blocks to make your model look like the figure.



5 From the Simulation menu, choose **Simulation parameters**.

6 Set the Simulation parameters as indicated in the following figure.



Simulation time

Start time: 0.0

Stop time: inf

Solver options

Type: Fixed-step

discrete (no continuous states)

Fixed step size: auto

Mode: Auto

Output options

Refine output

Refine factor: 1

7 Click **OK**.

8 In the model window, from the **Simulation** menu, choose **Start**.

The model simulation begins and the scope displays the three input signals. When you finish observing the running model, from the **Simulation** menu, choose **Stop**.

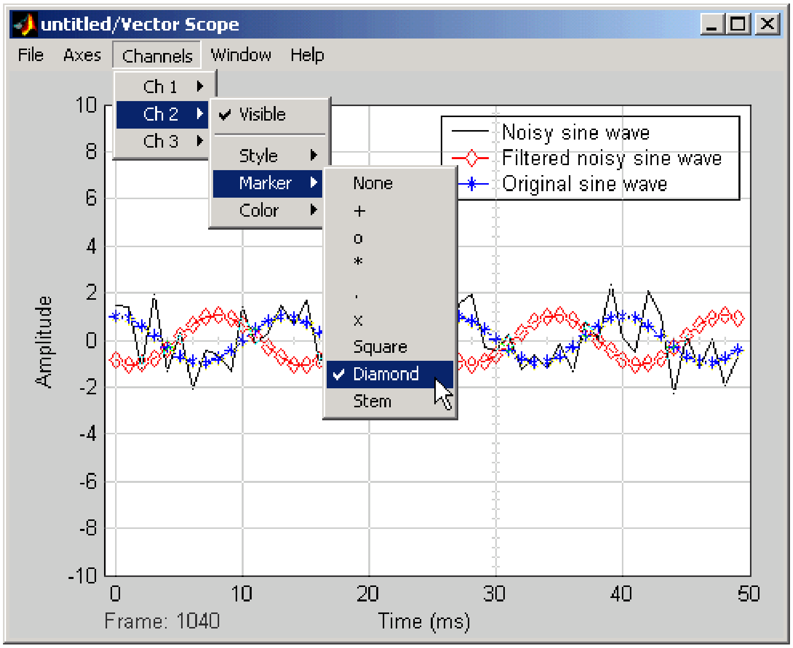
9 Double-click the Vector Scope block and select the **Show display properties** check box. Select the **Channel legend** check box and click **OK**. Next time you run the simulation, a legend appears in the Vector Scope window.

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

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

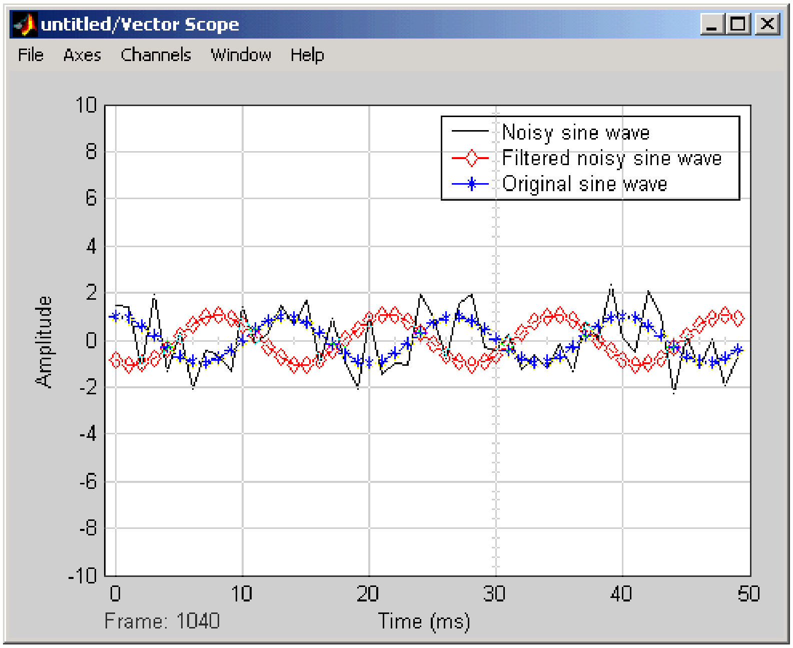
Point to **Ch 2** and set the **Style** to **-**, **Marker** to **Diamond**, and **Color** to **Red**.

Point to **Ch 3** and set the **Style** to **None**, **Marker** to *, and **Color** to **Blue**.



11 Rerun the simulation and compare the original sine wave, noisy sine wave, and filtered noisy sine wave in the **Vector Scope** display.

You can see that the lowpass filter filters out the high-frequency noise in the noisy sine wave.



In this example, Digital Filter Design blocks were used to design and implement lowpass and highpass filters. For information on another block capable of designing and implementing filters, see “Filter Realization Wizard” on page 3-26.

To learn how to save your filter designs, see the “Saving and Opening Filter Design Sessions” section of the Signal Processing Toolbox documentation. To learn how to import and export your filter designs, see the “Importing and Exporting Quantized Filters” section of the Filter Design Toolbox documentation.

Filter Realization Wizard

The Filter Realization Wizard is another DSP Blockset block that can be used to design and implement digital filters. You can use this tool to filter single-channel floating-point or fixed-point signals. Like the Digital Filter Design block, double-clicking a Filter Realization Wizard block opens FDATool. Unlike the Digital Filter Design block, the Filter Realization Wizard starts FDATool with the **Realize Model** panel selected. This panel is optimized for use with the DSP Blockset.

For more information, see the Filter Realization Wizard block reference page. For information on choosing between the Digital Filter Design block and the Filter Realization Wizard, see “Choosing Between Filter Design Blocks” on page 3-14.

This section includes the following topics:

- “Designing and Implementing a Fixed-Point Filter” on page 3-26 — Create a fixed-point filter with the Filter Realization Wizard

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

Designing and Implementing a Fixed-Point Filter

In this section, a tutorial guides you through creating a fixed-point filter with the Filter Realization Wizard. You will use the Filter Realization Wizard to remove noise from a signal. This tutorial has the following parts:

- “Part 1 — Creating a Signal with Added Noise” on page 3-27
- “Part 2 — Creating a Fixed-Point Filter with the Filter Realization Wizard” on page 3-29
- “Part 3 — Building a Model to Filter a Signal” on page 3-38
- “Part 4 — Looking at Filtering Results” on page 3-42

Part 1 — Creating a Signal with Added Noise

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

1 Type

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

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

2 Create a noise signal by typing

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

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

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

3 Add the noise to the original signal by typing

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

at the command line.

4 Scale the signal with noise by typing

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

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

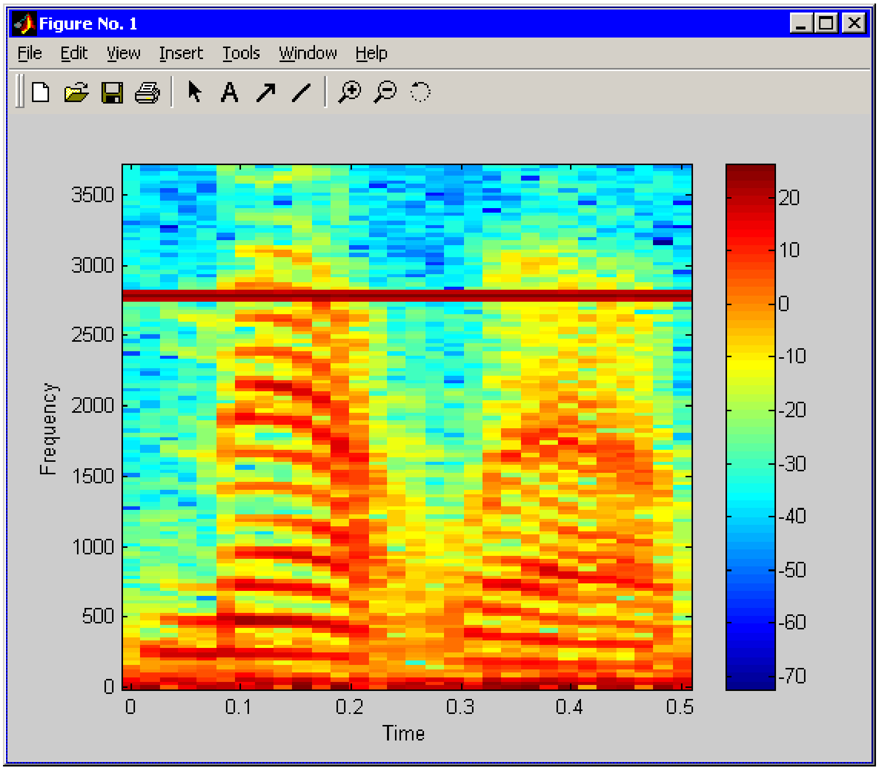
```
soundsc(u,Fs)
```

5 View the scaled signal with noise by typing

```
specgram(u,256,Fs);colorbar
```

at the command line.

The spectrogram will appear as follows.

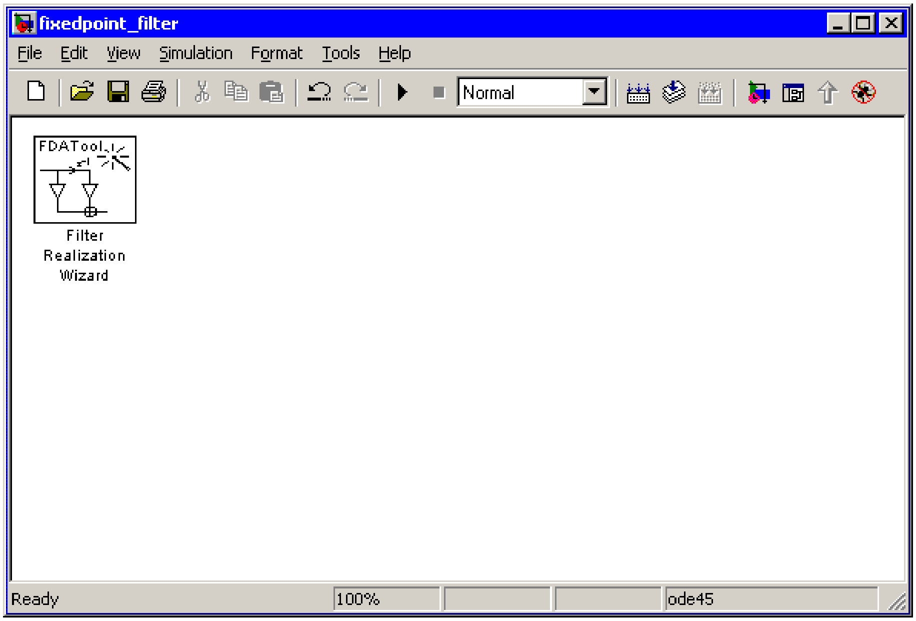


In the spectrogram, you can see the noise signal as a horizontal line at about 2800 Hz, which is equal to `3*Fs/8`.

Part 2 — Creating a Fixed-Point Filter with the Filter Realization Wizard

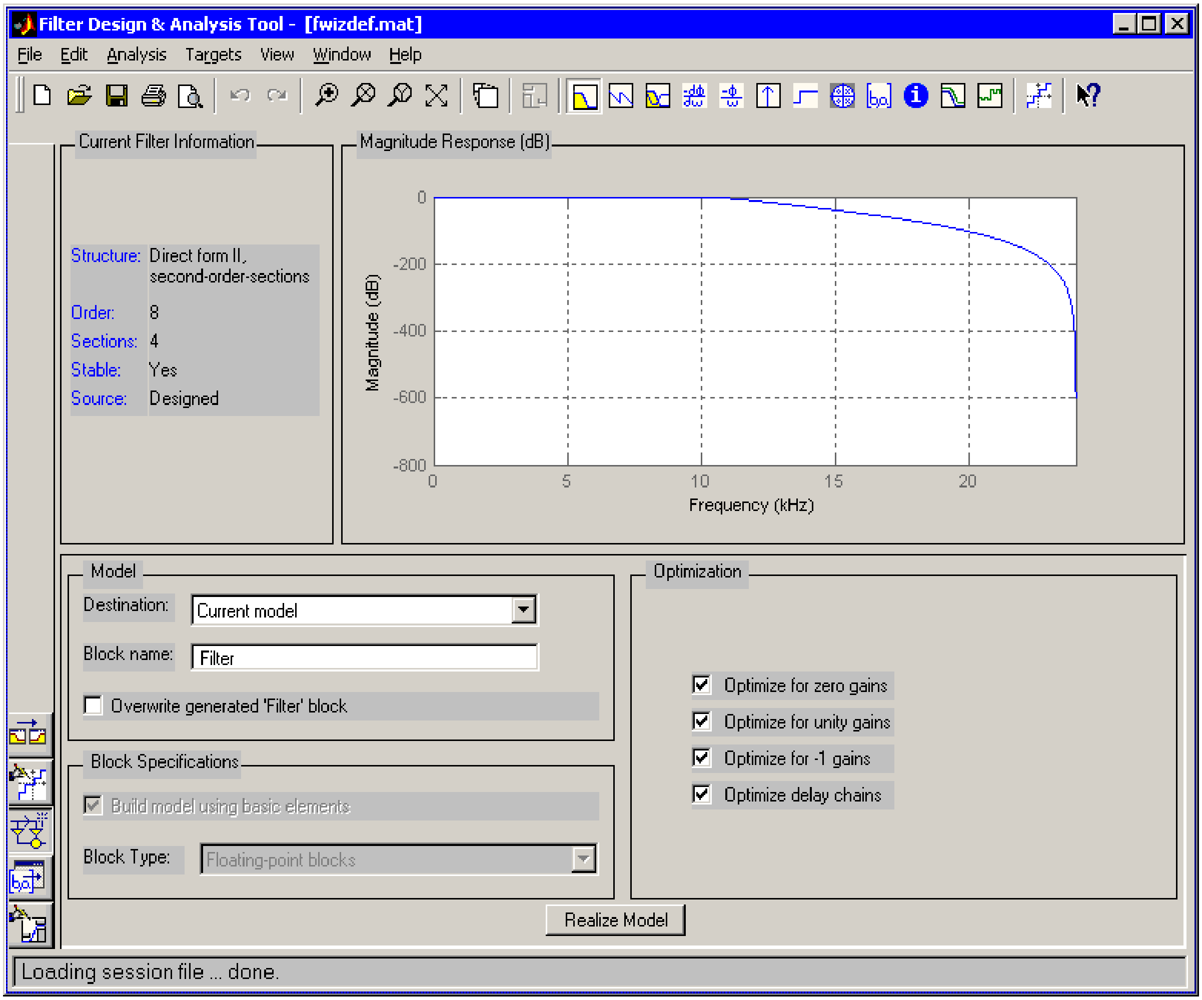
Next you will create a fixed-point filter using the Filter Realization Wizard. You will create a filter that reduces the effects of the noise on the signal.

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


fixedpoint_filter
File Edit View Simulation Format Tools Help
Normal
FDATool
Filter Realization Wizard
Ready
100%
ode45

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

7 Double-click the Filter Realization Wizard block in your model. The **Realize Model** panel of the Filter Design and Analysis Tool (FDATool) appears.



Structure: Direct form II, second-order-sections

Order: 8

Sections: 4

Stable: Yes

Source: Designed

Destination: Current model

Block name: Filter

□ Overwrite generated 'Filter' block

☑ Build model using basic elements

Block Type: Floating-point blocks

☑ Optimize for zero gains

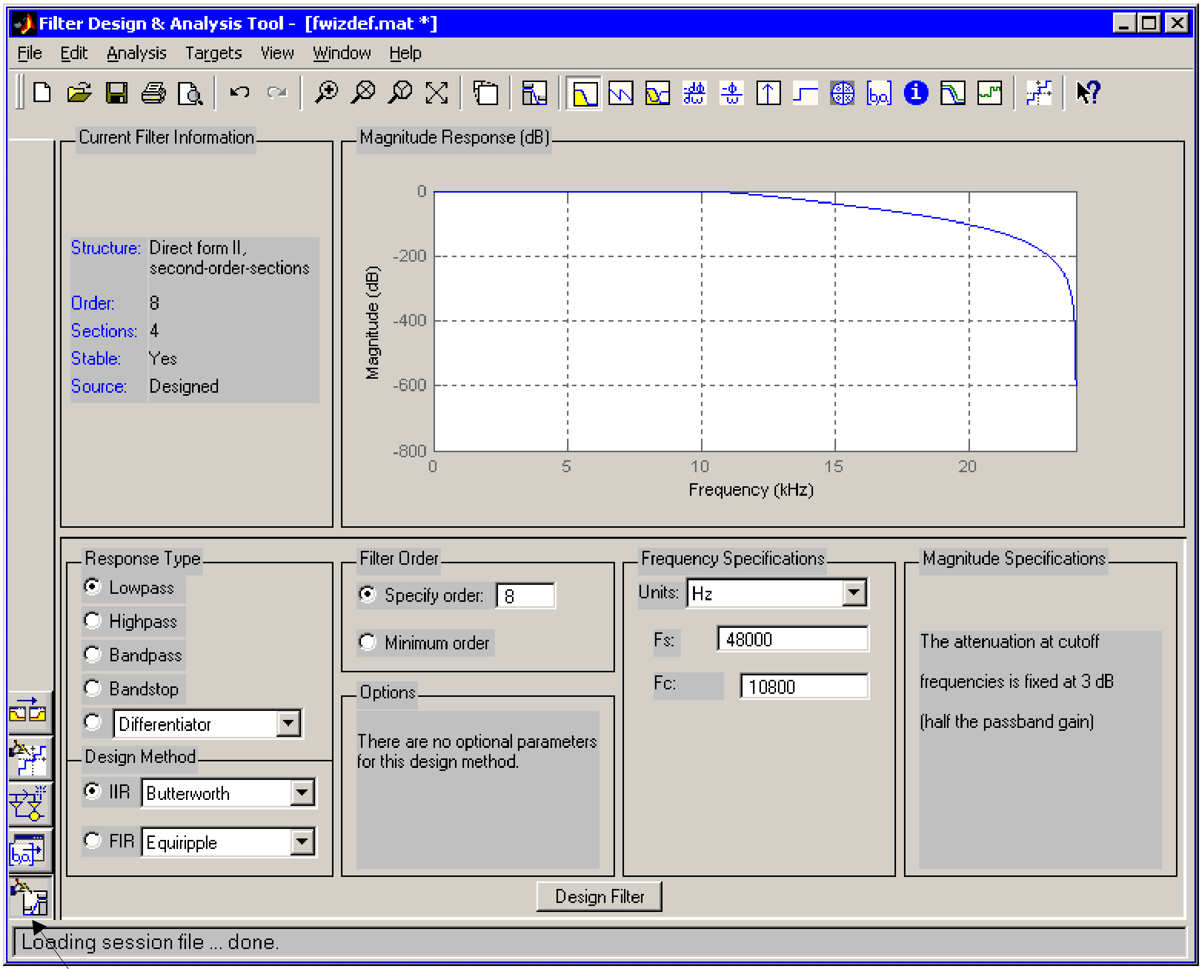
☑ Optimize for unity gains

☑ Optimize for -1 gains

☑ Optimize delay chains

Loading session file ... done.

8 Click the Design Filter button on the bottom left of FDATool. This brings forward the **Design Filter** panel of the tool.



Structure: Direct form II, second-order-sections

Order: 8

Sections: 4

Stable: Yes

Source: Designed

Response Type: ● Lowpass ○ Highpass ○ Bandpass ○ Bandstop ○ Differentiator

Design Method: ● IIR Butterworth ○ FIR Equiripple

Filter Order: ● Specify order: 8 ○ Minimum order

Options: There are no optional parameters for this design method.

Frequency Specifications: Units: Hz; Fs: 48000; Fc: 10800

Magnitude Specifications: The attenuation at cutoff frequencies is fixed at 3 dB (half the passband gain)

Loading session file ... done.

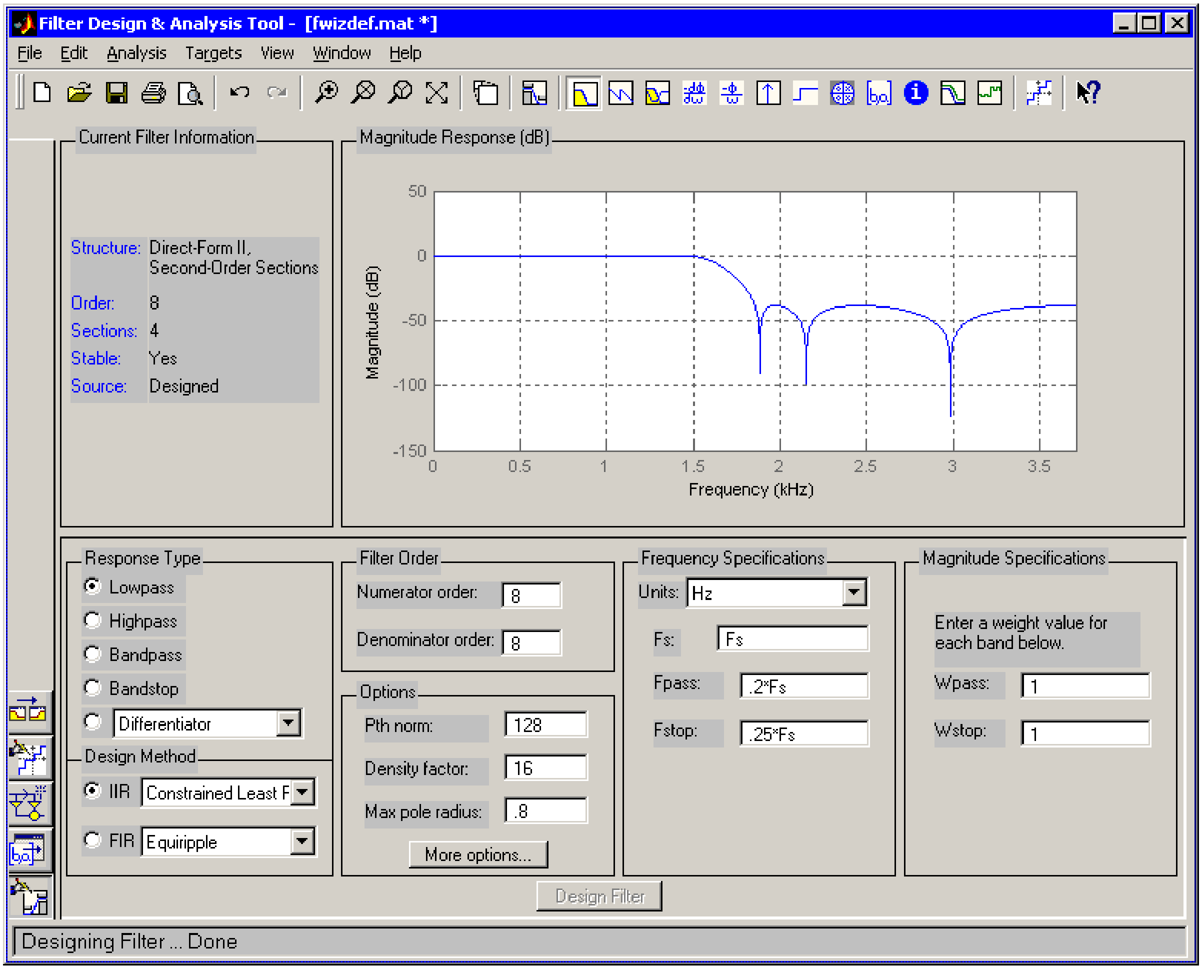
Design Filter button

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

- Set **Design Method** to `IIR` — `Constrained Least Pth-norm`
- Set **Fs** to `Fs`

- Set **Fpass** to `0.2*Fs`
- Set **Fstop** to `0.25*Fs`
- Set **Max pole radius** to `0.8`
- Click the **Design Filter** button

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



Current Filter Information

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

Response Type

● Lowpass
○ Highpass
○ Bandpass
○ Bandstop
○ Differentiator

Design Method

● IIR Constrained Least F
○ FIR Equiripple

Filter Order

Numerator order: 8
Denominator order: 8

Options

Pth norm: 128
Density factor: 16
Max pole radius: .8

Frequency Specifications

Units: Hz
Fs: Fs
Fpass: .2*Fs
Fstop: .25*Fs

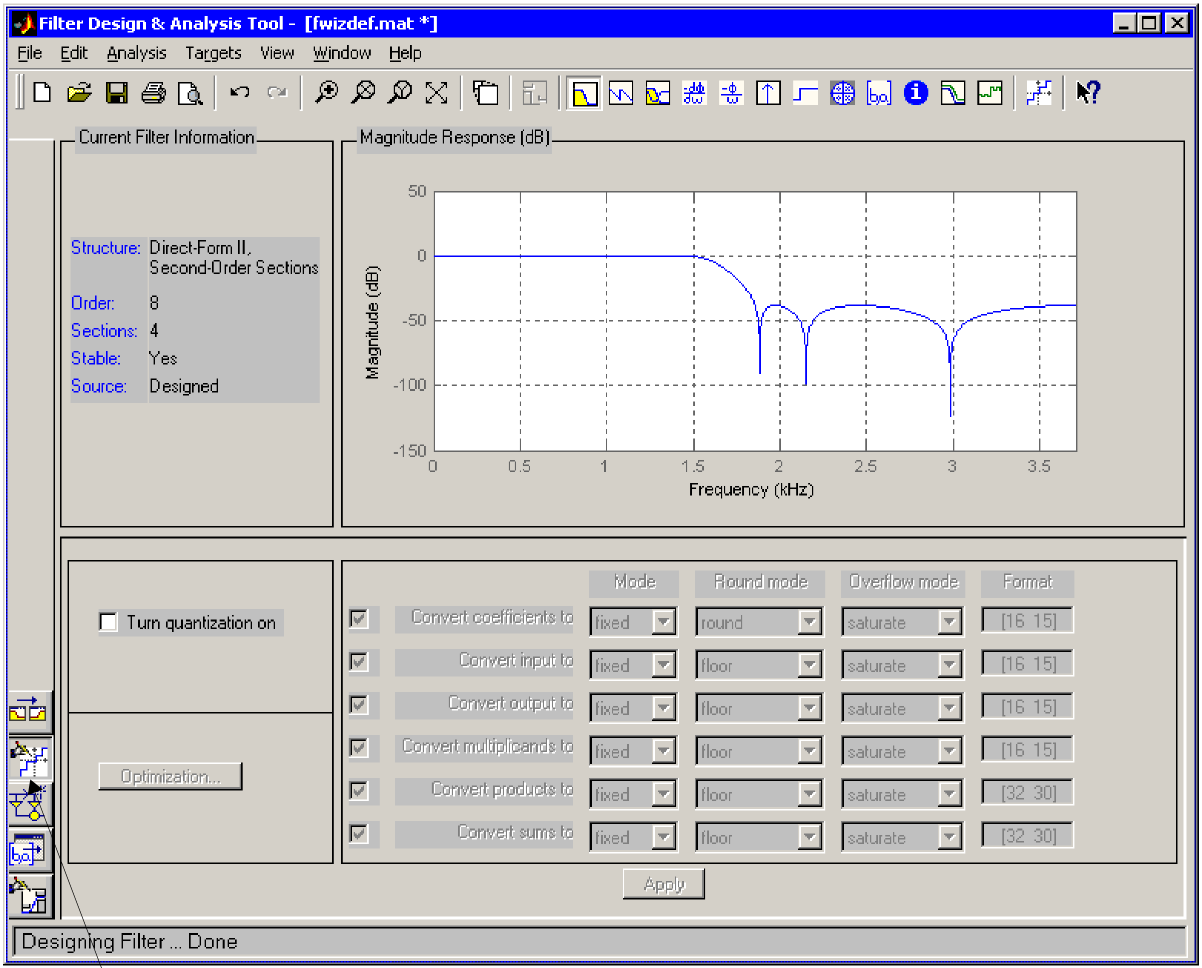
Magnitude Specifications

Enter a weight value for each band below.

Wpass: 1
Wstop: 1

Designing Filter ... Done

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



Filter Design & Analysis Tool - [fwizdef.mat *]

Current Filter Information

Structure: Direct-Form II, Second-Order Sections

Order: 8

Sections: 4

Stable: Yes

Source: Designed

□ Turn quantization on

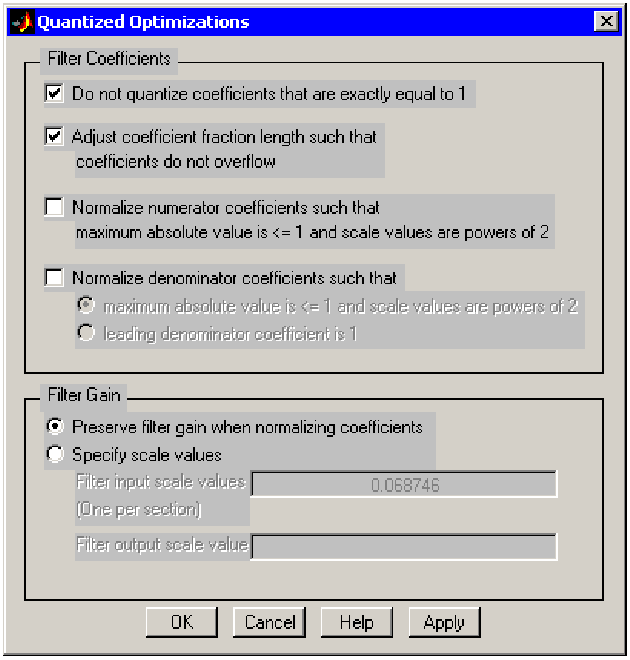
		Mode	Round mode	Overflow mode	Format
☑	Convert coefficients to	fixed	round	saturate	[16 15]
☑	Convert input to	fixed	floor	saturate	[16 15]
☑	Convert output to	fixed	floor	saturate	[16 15]
☑	Convert multiplicands to	fixed	floor	saturate	[16 15]
☑	Convert products to	fixed	floor	saturate	[32 30]
☑	Convert sums to	fixed	floor	saturate	[32 30]

Designing Filter ... Done

Set Quantization Parameters button

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

- Select the **Turn quantization on** check box.
- Click the **Optimization** button, which will open the **Quantized Optimizations** dialog.
- Select the **Do not quantize coefficients that are exactly equal to 1** and the **Adjust coefficient fraction length such that coefficients do not overflow** check boxes.



Quantized Optimizations

Filter Coefficients

☑ Do not quantize coefficients that are exactly equal to 1

☑ Adjust coefficient fraction length such that coefficients do not overflow

☐ Normalize numerator coefficients such that maximum absolute value is <= 1 and scale values are powers of 2

☐ Normalize denominator coefficients such that

● maximum absolute value is <= 1 and scale values are powers of 2

○ leading denominator coefficient is 1

Filter Gain

● Preserve filter gain when normalizing coefficients

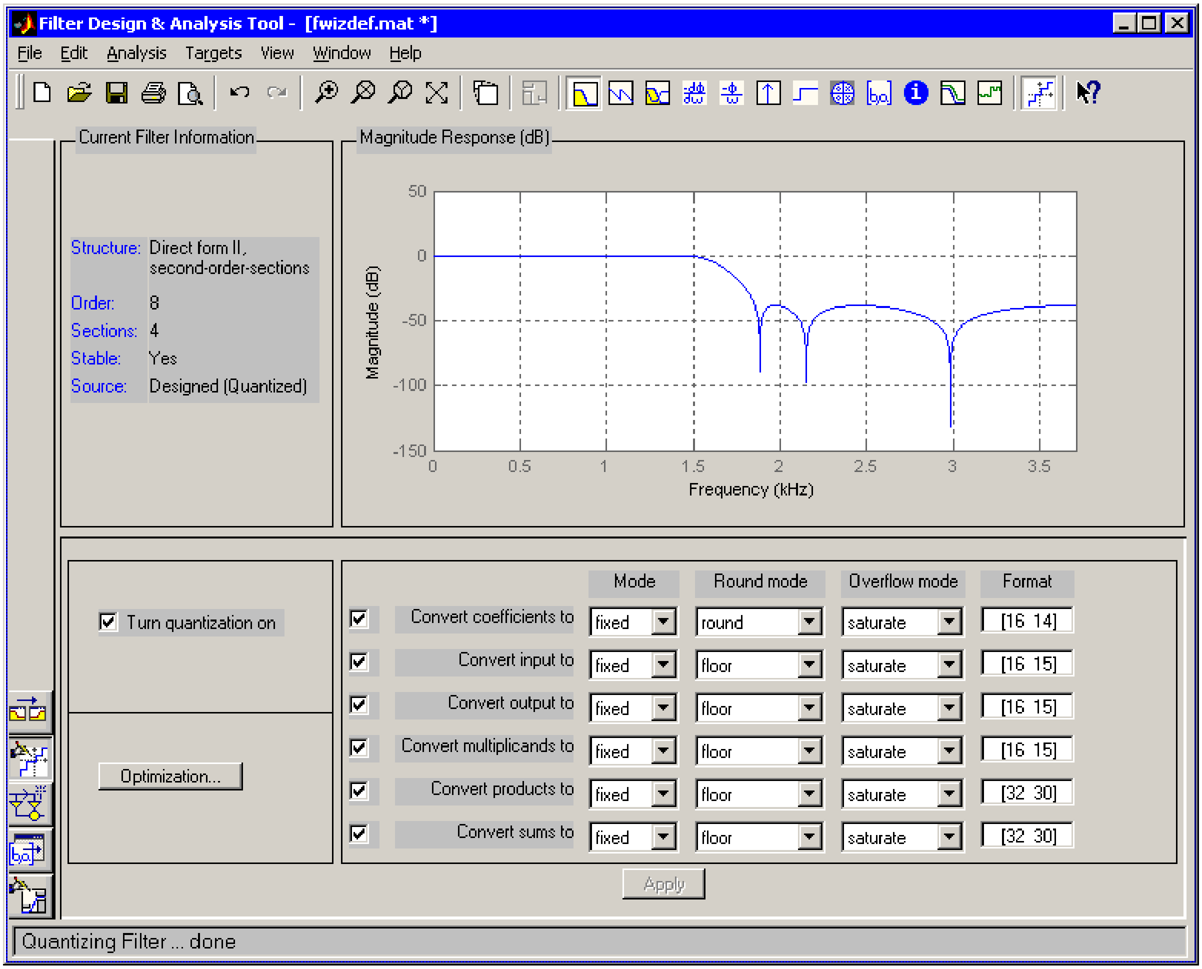
○ Specify scale values

Filter input scale values (One per section): 0.068746

Filter output scale value:

- Click **OK** in the **Quantized Optimizations** dialog.

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



Structure: Direct form II, second-order-sections

Order: 8

Sections: 4

Stable: Yes

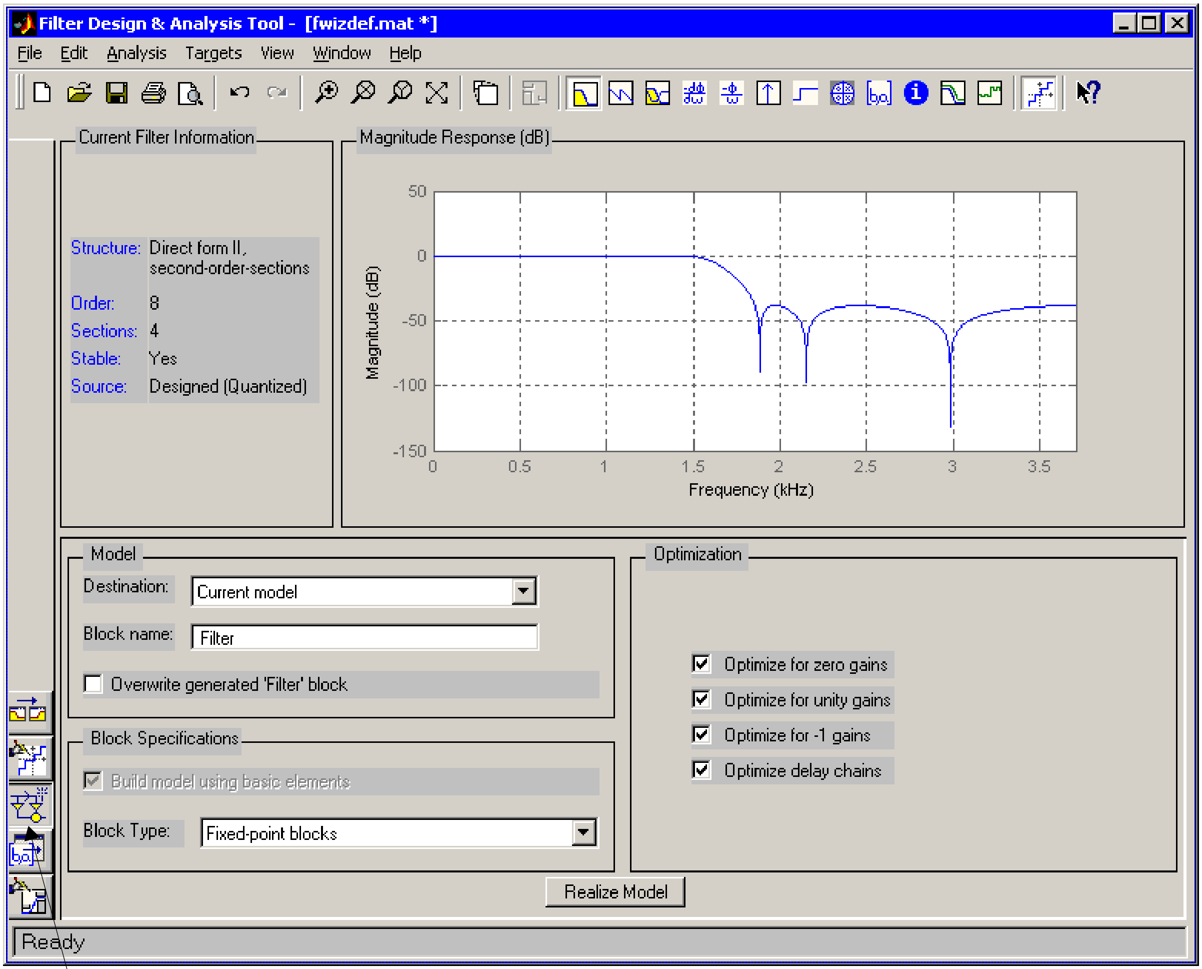
Source: Designed (Quantized)

☑ Turn quantization on

		Mode	Round mode	Overflow mode	Format
☑	Convert coefficients to	fixed	round	saturate	[16 14]
☑	Convert input to	fixed	floor	saturate	[16 15]
☑	Convert output to	fixed	floor	saturate	[16 15]
☑	Convert multiplicands to	fixed	floor	saturate	[16 15]
☑	Convert products to	fixed	floor	saturate	[32 30]
☑	Convert sums to	fixed	floor	saturate	[32 30]

Quantizing Filter ... done

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



Current Filter Information

Structure: Direct form II, second-order-sections

Order: 8

Sections: 4

Stable: Yes

Source: Designed (Quantized)

Model

Destination: Current model

Block name: Filter

☐ Overwrite generated 'Filter' block

Block Specifications

☑ Build model using basic elements

Block Type: Fixed-point blocks

Optimization

☑ Optimize for zero gains

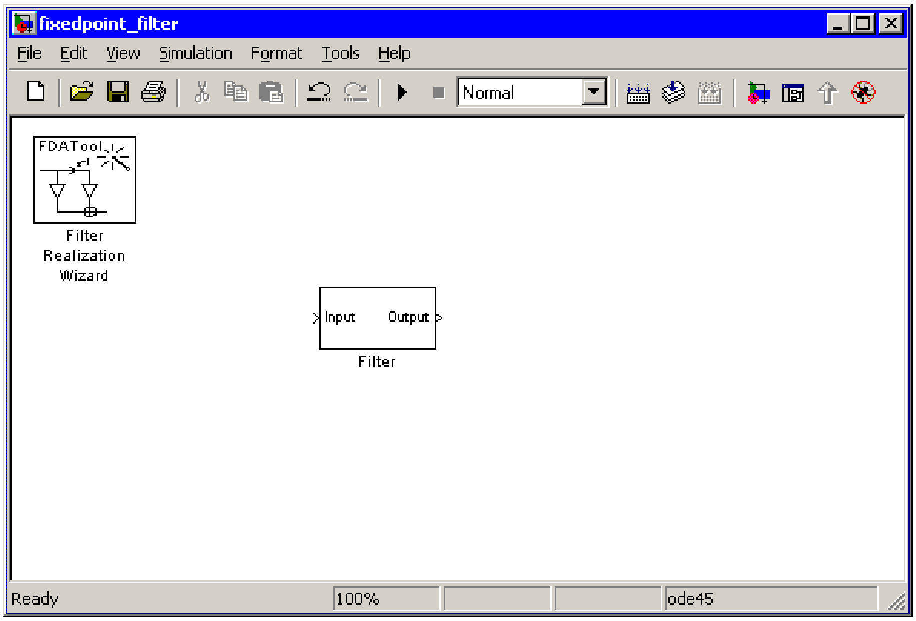
☑ Optimize for unity gains

☑ Optimize for -1 gains

☑ Optimize delay chains

Realize Model button

13 Click the **Realize Model** button. A block for the new filter appears in your model.

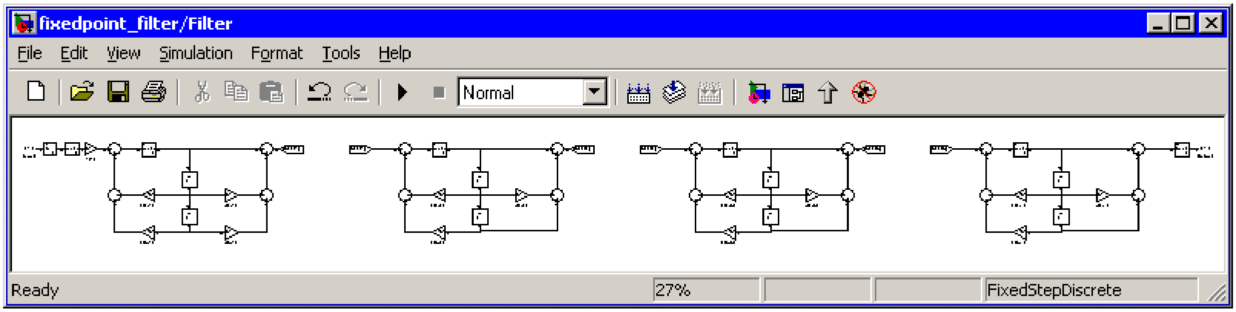


fixedpoint_filter
File Edit View Simulation Format Tools Help
Normal
FDATool
Filter Realization Wizard
Input
Output
Filter
Ready
100%
ode45

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

---

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

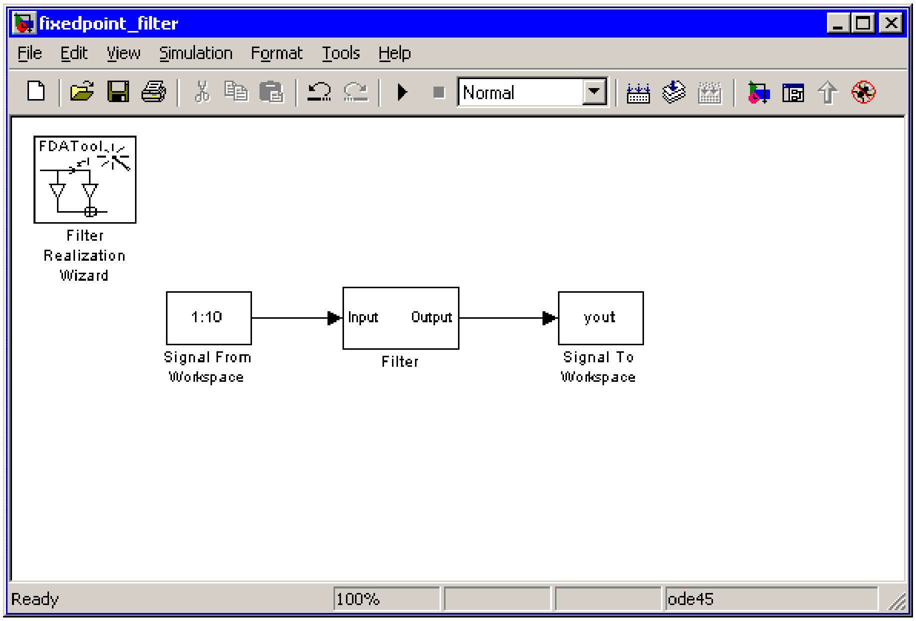


Part 3 — Building a Model to Filter a Signal

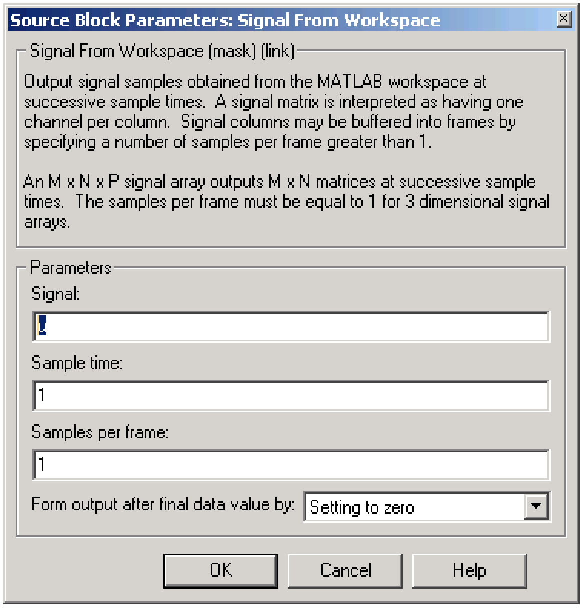
In this section of the tutorial, you will build and run a model with the filter you just designed, in order to filter the noise from your signal.

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

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



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



Source Block Parameters: Signal From Workspace

Signal From Workspace (mask) (link)

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: u

Sample time: 1

Samples per frame: 1

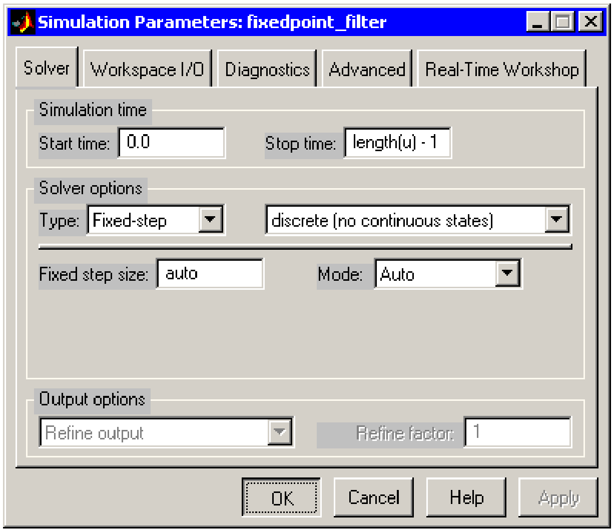
Form output after final data value by: Setting to zero

18 Click the **OK** button.

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

- Set **Stop time** to `length(u) - 1`.
- Set **Type** to `Fixed-step`.

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



Simulation Parameters: fixedpoint_filter

Simulation time

Start time: 0.0

Stop time: length(u) - 1

Solver options

Type: Fixed-step

discrete (no continuous states)

Fixed step size: auto

Mode: Auto

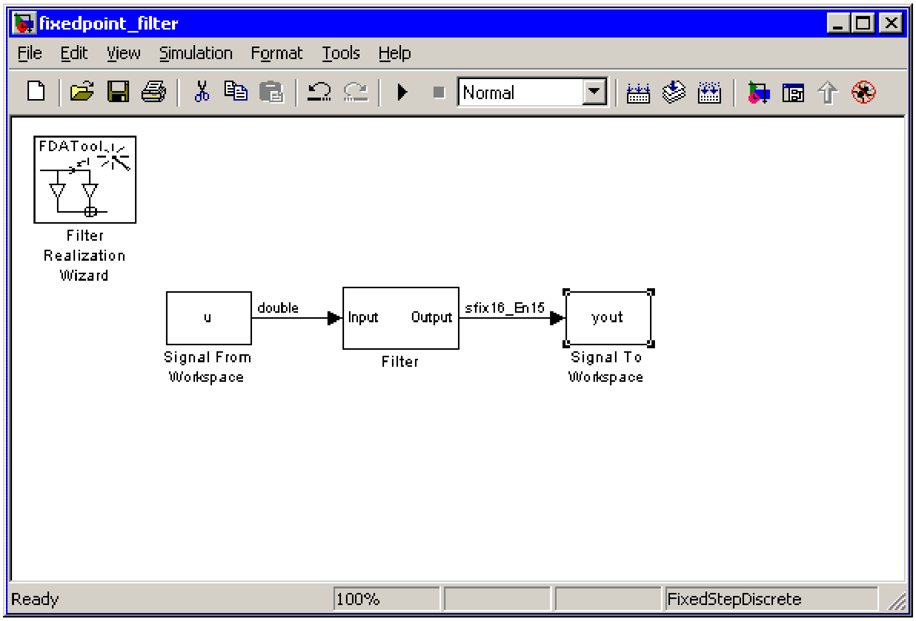
Output options

Refine output

Refine factor: 1

20 Click the **OK** button.

21 Run the model.



If you have `Port data types` selected in the **Format** menu, you can see that a signal of type `double` is entering your Filter block, and a signal of type `sfix16_En15` is exiting your Filter block.

Part 4 — Looking at Filtering Results

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

22 Type

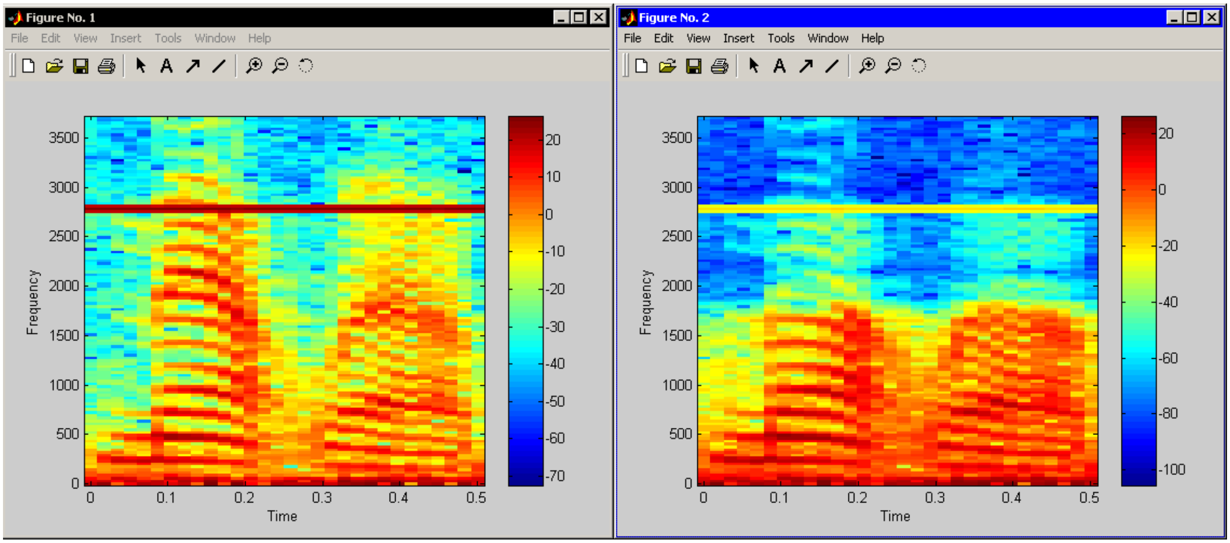
```
soundsc(yout,Fs)
```

at the command line to hear the output of the filter. You should hear a voice say “MATLAB.” The noise portion of the signal should be close to inaudible.

23 Type

```
figure
specgram(yout, 256, Fs);colorbar
```

at the command line. You can compare the input and output signals side-by-side.



From the colorbars at the side of each spectrogram, you can see that the noise has been reduced by about 40 dB.

Analog Filter Design Block

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

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

The table parameters are

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

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

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

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

- The elliptic design uses the toolbox function `ellip`.

The Analog Filter Design block is built on the filter design capabilities of the Signal Processing Toolbox. For more information on the filter design algorithms, see 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 (no continuous states)` option is selected in the **Solver** panel of the **Simulation Parameters** dialog box. Select one of the continuous solvers (such as `ode4`) instead.

Adaptive Filters

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

This section includes the following topics:

- "Creating an Acoustic Environment" on page 3-46 — Build a subsystem that models white noise and colored noise added to an input signal
- "Creating an Adaptive Filter" on page 3-47 — Build an adaptive filter using an LMS Filter block
- "Customizing an Adaptive Filter" on page 3-53 — Modify your adaptive filter and change its parameters during simulation
- "Adaptive Filtering Demos" on page 3-60 — Explore the adaptive filtering demos in the DSP Blockset

Creating an Acoustic Environment

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

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

   The DSP Blockset Acoustic Noise Cancellation demo opens.

2. Copy and paste the subsystem called Acoustic Environment into a new model file.

3. Double-click the Acoustic Environment subsystem.

   Gaussian noise is used to create the signal sent to the Exterior Mic output port. If the input to the Filter port changes from 0 to 1, the Digital Filter block changes from a lowpass filter to a bandpass filter. The filtered noise output from the Digital Filter block is added to signal coming from a `.wav` file to produce the signal sent to the Pilot's Mic output port.

You have now created an acoustic environment. In the following topics, you use this acoustic environment to produce a model capable of adaptive noise cancellation. Your next task is to create an adaptive filter. See “Creating an Adaptive Filter” on page 3-47.

Creating an Adaptive Filter

In the previous topic, “Creating an Acoustic Environment” on page 3-46, you created a system that produced two output signals. The signal output at the Exterior Mic port is composed of white noise. The signal output at the Pilot’s Mic port is composed of colored noise added to a signal from a `.wav` file. In this topic, you create an adaptive filter to remove the noise from the Pilot’s Mic signal. This topic assumes that you are working on a Windows operating system and have completed the procedures described in “Creating an Acoustic Environment” on page 3-46:

1. From the Adaptive Filters library, click-and-drag an LMS Filter block into the model that contains the Acoustic Environment subsystem.

2. Double-click the LMS Filter block.

   The **Block Parameters: LMS Filter** dialog box opens.

3. From the **Algorithm** list, select `Normalized LMS`.

   The block uses the normalized LMS algorithm to calculate the filter coefficients.

4. For the **Filter length** parameter, enter `40` to specify an adaptive filter with forty coefficients.

5. For the **Step-size (mu)** parameter, enter `0.002`.

6. For the **Leakage factor (0 to 1)** parameter, enter `1`.

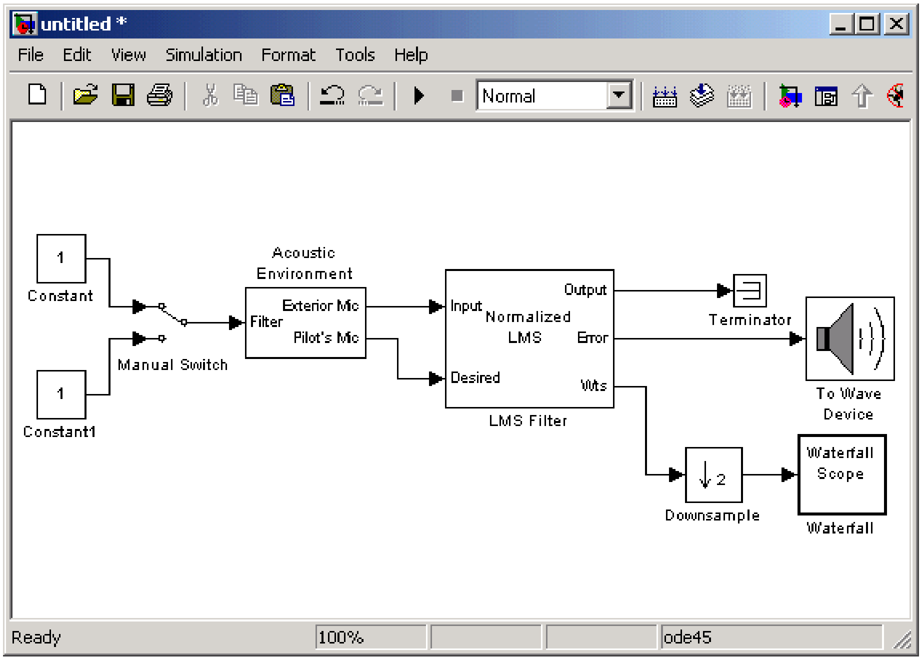
   Setting the **Leakage factor (0 to 1)** parameter to `1` means that the current filter coefficient values depend on the filter’s initial conditions and all of the previous input values.

7. Click **OK**.

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

Block	**Library**	**Quantity**
Constant	Simulink/Sources	2
Manual Switch	Simulink/Signal Routing	1
Terminator	Simulink/Sinks	1
To Wave Device	Platform Specific I/O/ Windows	1
Downsample	Signal Operations	1
Waterfall Scope	DSP Sinks	1

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

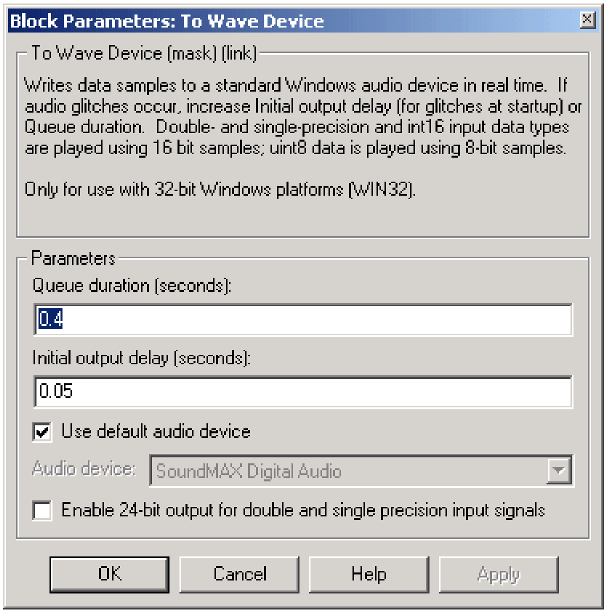


10 Double-click the Constant block.

11 Set the **Constant value** parameter to `0`

12 Click **OK**.

13 Double-click the To Wave Device block and set the parameters as follows.



Block Parameters: To Wave Device

To Wave Device (mask) (link)

Writes data samples to a standard Windows audio device in real time. If audio glitches occur, increase Initial output delay (for glitches at startup) or Queue duration. Double- and single-precision and int16 input data types are played using 16 bit samples; uint8 data is played using 8-bit samples.

Only for use with 32-bit Windows platforms (WIN32).

Parameters

Queue duration (seconds): 0.4

Initial output delay (seconds): 0.05

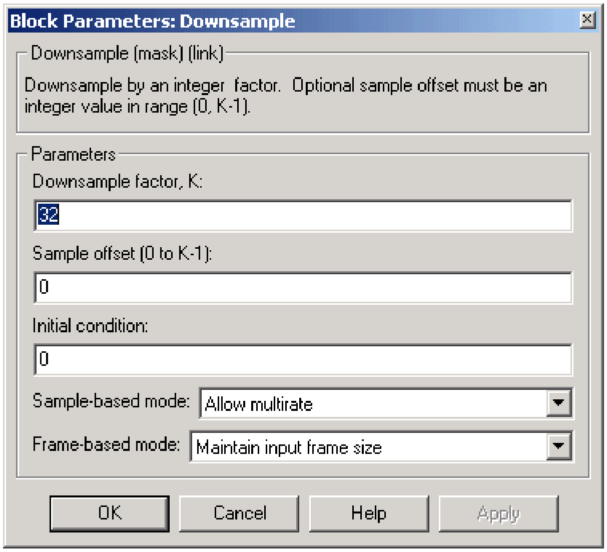
☑ Use default audio device

Audio device: SoundMAX Digital Audio

□ Enable 24-bit output for double and single precision input signals

Click **OK**.

14 Double-click the Downsample block. Set the parameters as shown in the following figure.



Block Parameters: Downsample

Downsample (mask) (link)

Downsample by an integer factor. Optional sample offset must be an integer value in range [0, K-1].

Parameters

Downsample factor, K: 32

Sample offset (0 to K-1): 0

Initial condition: 0

Sample-based mode: Allow multirate

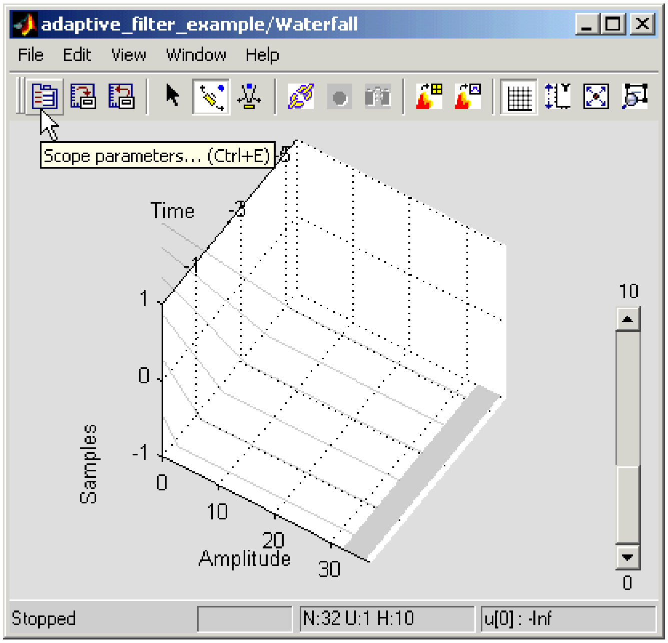
Frame-based mode: Maintain input frame size

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

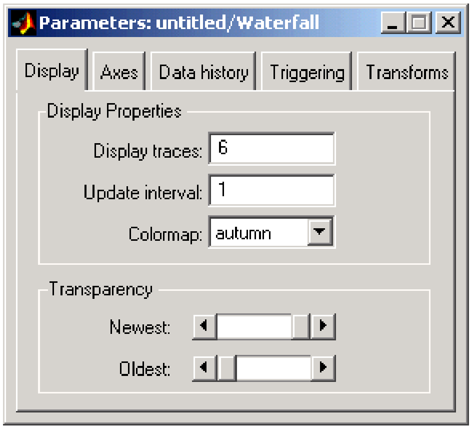
15 Click **OK**.

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

17 Click on the Scope parameters button.



The **Parameters** window opens.


Parameters: untitled/Waterfall

Display Properties

Display traces: 6

Update interval: 1

Colormap: autumn

Transparency

Newest:

Oldest:

18 Click on the **Axes** tab. For the **Y Min** parameter, enter `-0.188`. For the **Y Max** parameter, enter `0.179`.

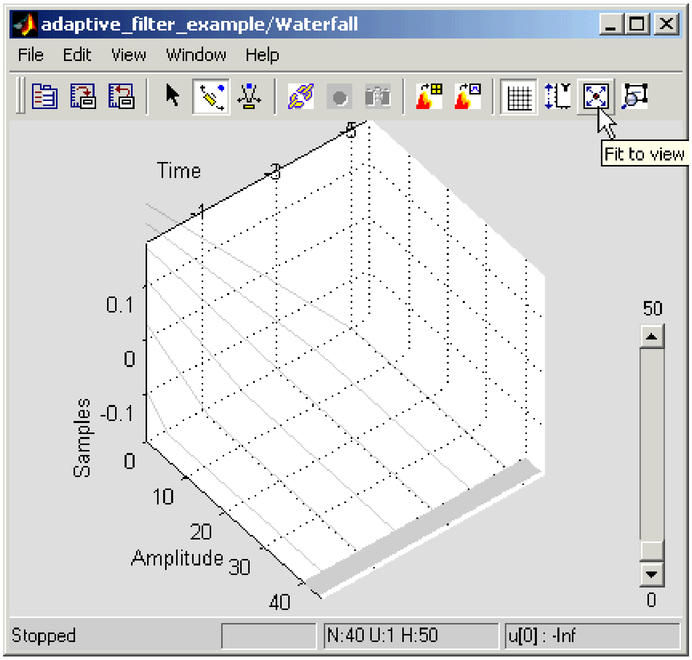
19 Click on the **Data history** tab. For the **History traces** parameter, enter `50`. From the **Data logging** list, choose `All visible`.

20 Close the **Parameters** window leaving all other parameters at their default values.

You might need to adjust the axes in the **Waterfall** scope window in order to view the plots.

21 Click on the **Fit to view** button in the **Waterfall** scope window.

22 Then click-and-drag the axes until they resemble the following figure.



Stopped | N:40 U:1 H:50 | u[0] : -Inf

23 In the model window, from the **Simulation** menu, select **Simulation parameters**.

   The **Simulation parameters** dialog box opens.

24 Click on the **Solver** tab.

25 Set the **Stop time** to `inf`.

26 From the **Type** list, select `Fixed-step` and `discrete (no continuous states)`.

27 Click **OK**.

28 Run the simulation and view the results in the **Waterfall** scope window. You can also listen to the simulation using the speakers attached to your computer.

29 Experiment with changing the Manual Switch so that the input to the Acoustic Environment subsystem is either 0 or 1.

   When the value is 0, the Gaussian noise in the signal is being filtered by a lowpass filter. When the value is 1, the noise is being filtered by a bandpass filter. The adaptive filter can remove the noise in both cases.

You have now created a model capable of adaptive noise cancellation. The adaptive filter in your model is able to filter out both low frequency noise and noise within a frequency range. In the following topic, "Customizing an Adaptive Filter" on page 3-53, you modify the LMS Filter block and change its parameters during simulation.

Customizing an Adaptive Filter

In the previous topic, "Creating an Adaptive Filter" on page 3-47, you created an adaptive filter and used it to remove the noise generated by the Acoustic Environment subsystem. In this topic, you modify the adaptive filter and adjust its parameters during simulation. This topic assumes that you are working on a Windows operating system and that you completed the procedures discussed in "Creating an Adaptive Filter" on page 3-47:

1 Double-click the LMS filter block.

2 From the **Specify step-size via** list, choose `Input port`.

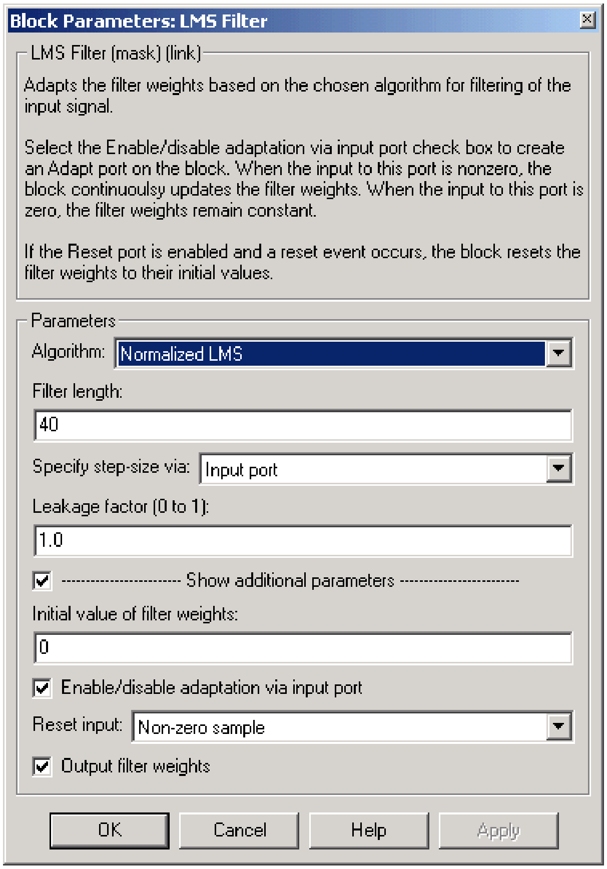
3 Select the **Show additional parameters** check box.

4 For the **Initial value of filter weights** parameter, enter `0` to set the initial filter coefficients to 0.

5 Select the **Enable/disable adaptation via input port** check box.

6 From the **Reset input** list, select `Non-zero sample`.

The **Block Parameters: LMS Filter** dialog box should now look similar to the following figure.



Block Parameters: LMS Filter

LMS Filter (mask) (link)

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

Select the Enable/disable adaptation via input port check box to create an Adapt port on the block. When the input to this port is nonzero, the block continuoulsy updates the filter weights. When the input to this port is zero, the filter weights remain constant.

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

Parameters

Algorithm: Normalized LMS

Filter length: 40

Specify step-size via: Input port

Leakage factor (0 to 1): 1.0

☑ ---------------------- Show additional parameters -----------------------

Initial value of filter weights: 0

☑ Enable/disable adaptation via input port

Reset input: Non-zero sample

☑ Output filter weights

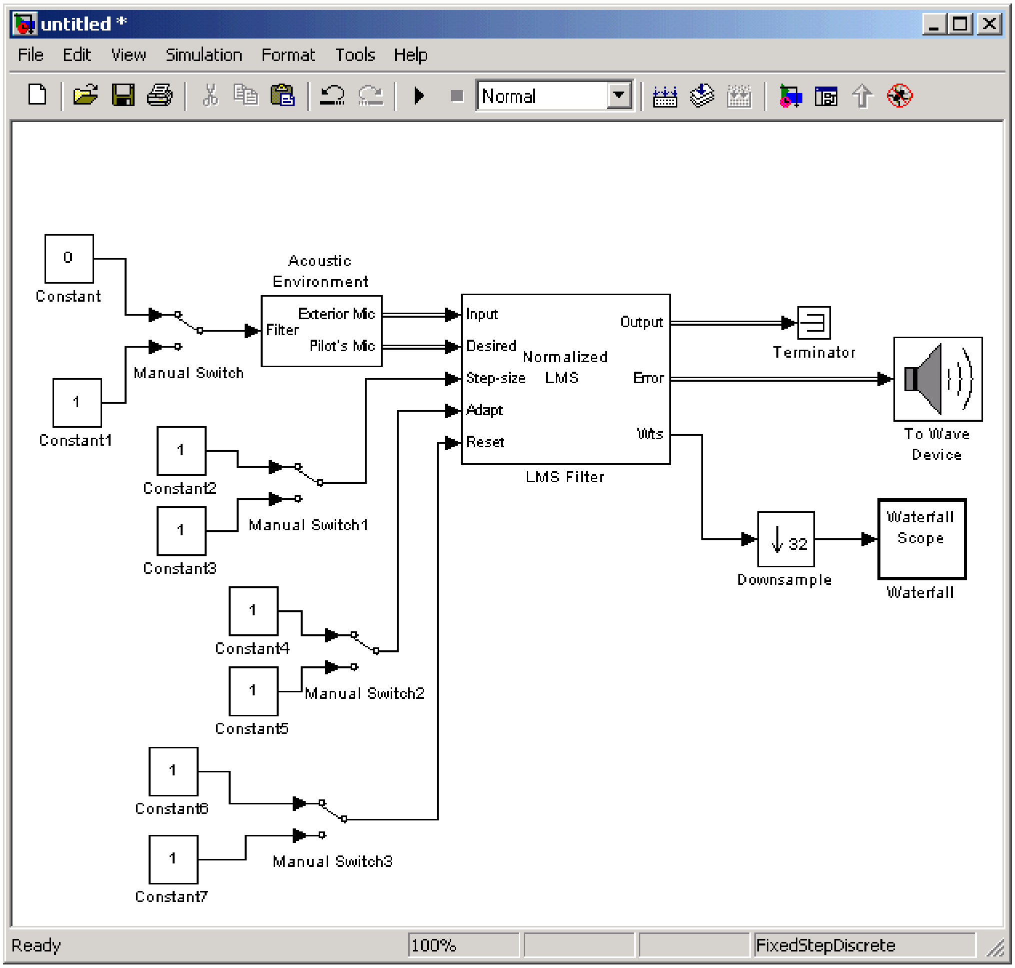
7 Click **OK**.

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

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

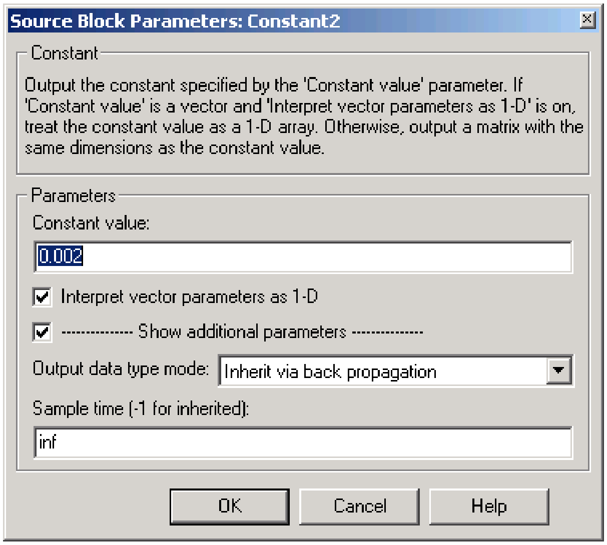
Block	Library	Quantity
Constant	Simulink/Sources	6
Manual Switch	Simulink/Signal Routing	3

9 Connect the blocks as shown in the following figure.


untitled *
File
Edit
View
Simulation
Format
Tools
Help
Normal
0
Constant
1
Constant1
Manual Switch
Acoustic Environment
Exterior Mic
Filter
Pilot's Mic
1
Constant2
1
Constant3
Manual Switch1
1
Constant4
1
Constant5
Manual Switch2
1
Constant6
1
Constant7
Manual Switch3
Input
Desired
Step-size
Adapt
Reset
Normalized LMS
Output
Error
Wts
LMS Filter
Terminator
To Wave Device
32
Downsample
Waterfall Scope
Waterfall
Ready
100%
FixedStepDiscrete

10 Double-click the Constant2 block. Set the parameters as shown in the following figure.

Note Opening a dialog box for a source block causes the simulation to pause. While the simulation is paused, you can edit the parameter values. You must close the dialog box to have the changes take effect and allow the simulation to continue.



Source Block Parameters: Constant2

Constant

Output the constant specified by the 'Constant value' parameter. If 'Constant value' is a vector and 'Interpret vector parameters as 1-D' is on, treat the constant value as a 1-D array. Otherwise, output a matrix with the same dimensions as the constant value.

Parameters

Constant value: 0.002

☑ Interpret vector parameters as 1-D

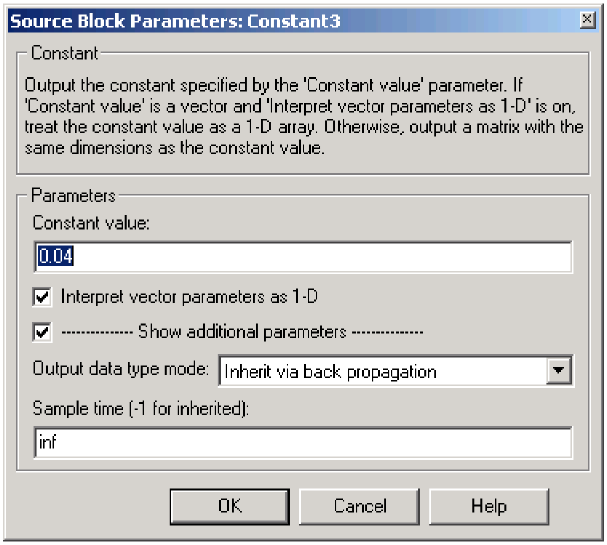
☑ -------------- Show additional parameters --------------

Output data type mode: Inherit via back propagation

Sample time (-1 for inherited): inf

11 Click **OK**.

12 Double-click the Constant3 block. Set the parameters as shown below.



Source Block Parameters: Constant3

Constant

Output the constant specified by the 'Constant value' parameter. If 'Constant value' is a vector and 'Interpret vector parameters as 1-D' is on, treat the constant value as a 1-D array. Otherwise, output a matrix with the same dimensions as the constant value.

Parameters

Constant value: 0.04

☑ Interpret vector parameters as 1-D

☑ ------------- Show additional parameters -------------

Output data type mode: Inherit via back propagation

Sample time (-1 for inherited): inf

13 Click **OK**.

14 Double-click the Constant4 block and set the **Constant value** parameter to 0.

15 Click **OK**.

16 Double-click the Constant6 block and set the **Constant value** parameter to 0.

17 Click **OK**.

18 In the model window, from the **Format** menu, select **Wide nonscalar lines** and **Signal dimensions**.

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

20 Run the simulation and view the results in the **Waterfall** scope window. You can also listen to the simulation using the speakers attached to your computer.

21 Double-click the Manual Switch so that the input to the Acoustic Environment subsystem is 1. Then, double-click Manual Switch2 so that the input to the Adapt port to 0.

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

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

   The LMS Filter block updates the coefficients.

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

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

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

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

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

For more information about adaptive filters, see the following block reference pages: “LMS Filter” on page 7-453, “RLS Filter” on page 7-640, “Block LMS Filter” on page 7-46, and “Fast Block LMS Filter” on page 7-295.

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.

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`

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.

Multirate Filters

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

This section includes the following topic:

- "Multirate Filtering Demos" on page 3-61 — Explore the multirate filtering demos in the DSP Blockset

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.

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

Opening Demos. To open the multirate filter demos, click on the links in the following table in the MATLAB Help browser (not in a Web browser), or type the demo names provided in the table at the MATLAB command line. To access all DSP Blockset demos, type `demo blockset dsp` at the MATLAB command line.

4

Transforms

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

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 4-3

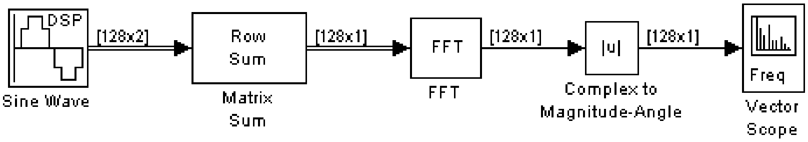
The first example loosely follows the example in the "Discrete Fourier Transform" section of the Signal Processing Toolbox documentation, where you can also find additional background information on these transform operations.

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.

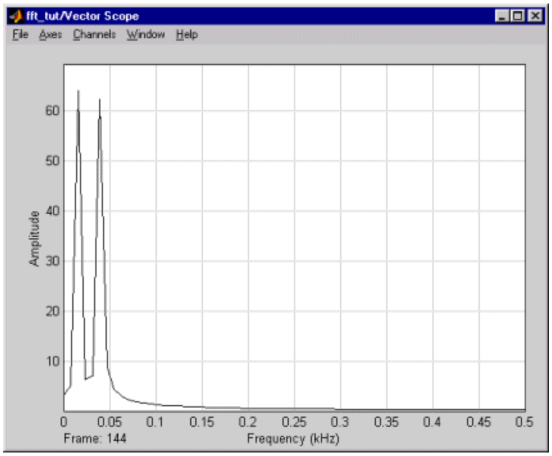


To build the model, make the following parameter settings:

- 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.

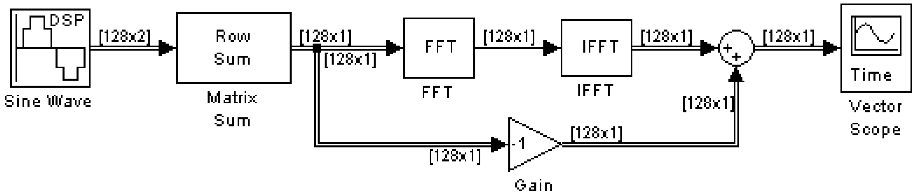
Other blocks that compute the FFT internally are the blocks in the Power Spectrum Estimation library. See “Power Spectrum Estimation” on page 5-5 for more information about these blocks.

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

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

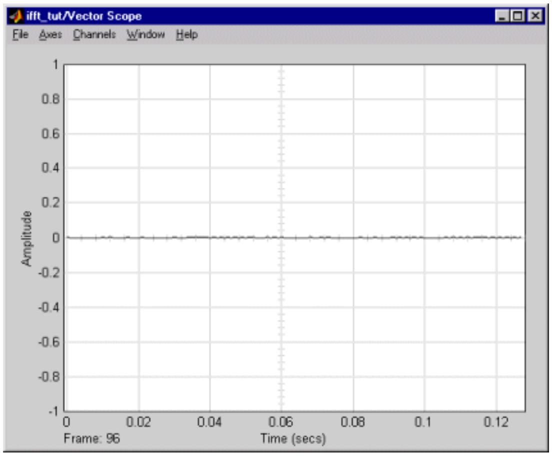
which is transformed to the frequency domain using an FFT block. The frequency-domain signal is then immediately transformed back to the time domain by the IFFT block, and the difference between the original time-domain signal and transformed time-domain signal is plotted on the scope.



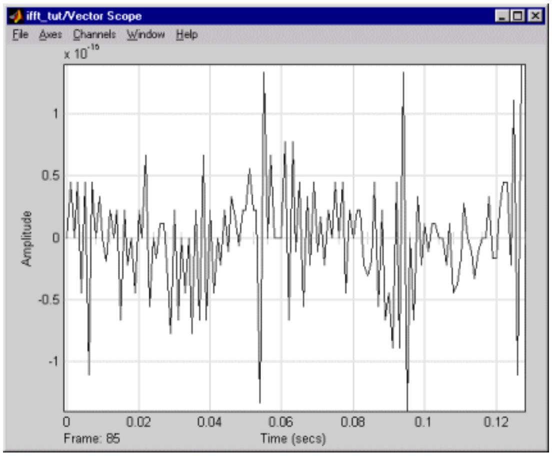
To build the model, make the following parameter settings (leave unlisted parameters in their default settings):

- 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}.



5

Statistics, Estimation, and Linear Algebra

This chapter describes several standard operations involved in simulating DSP models.

Statistics

The Statistics library provides fundamental statistical operations such as minimum, maximum, mean, variance, and standard deviation. Most blocks in the Statistics library support two types of operations:

- Basic operations
- Running operations

The blocks listed below toggle between basic and running modes using the **Running** check box in the parameter dialog box:

- Histogram
- Mean
- RMS
- Standard Deviation
- Variance

An unselected **Running** box means that the block is operating in basic mode, while a selected **Running** box means that the block is operating in running mode.

The Maximum and Minimum blocks are slightly different from the blocks above, and provide a **Mode** parameter in the block dialog box to select the type of operation. The `Value and Index`, `Value`, and `Index` options in the **Mode** menu all specify basic operation, in each case enabling a different set of output ports on the block. The `Running` option in the **Mode** menu selects running operation.

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

- “Basic Operations” on page 5-2
- “Running Operations” on page 5-4

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

Basic Operations

A *basic operation* is one that processes each input independently of previous and subsequent inputs. For example, in basic mode (with `Value and Index` selected, for example) the Maximum block finds the maximum value in each column of the current input, and returns this result at the top output (`Val`). Each consecutive `Val` output therefore has the same number of columns as the input, but only one row. Furthermore, the values in a given output only depend on the values in the corresponding input. The block repeats this operation for each successive input.

This type of operation is exactly equivalent to the MATLAB command

```
val = max(u)        % Equivalent MATLAB code
```

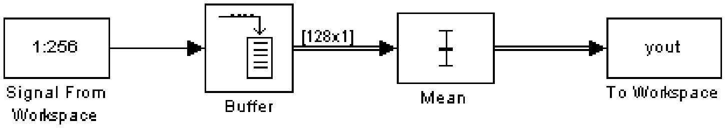
which computes the maximum of each column in input `u`.

The next section 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_o) parameter determines the size of the window. The **Buffer overlap** (L) parameter defines the "slide factor" for the window. At each sample instant, the window slides by M_o-L points. The **Buffer overlap** is often M_o-1 (the same as the Delay Line block), so that a new statistic is computed for every new signal sample.

To build the model, make the following settings:

- In the Signal From Workspace block, set:
  - **Signal** = `1:256`
  - **Sample time** = `0.1`
  - **Samples per frame** = `1`

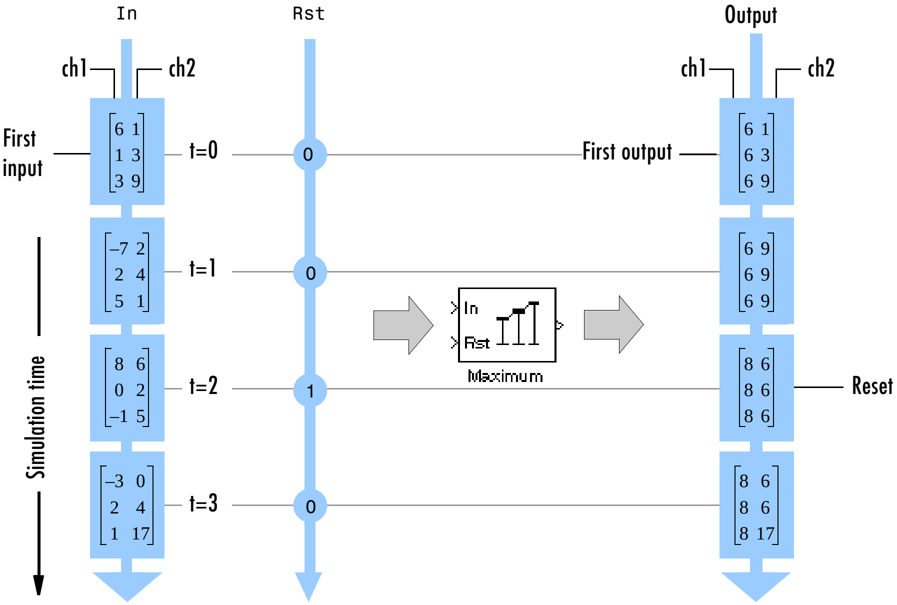
- In the Buffer block, set:
  - **Output buffer size (per channel)** = 128
  - **Buffer overlap** = 127

Running Operations

A *running operation* is one that processes successive sample-based or frame-based inputs, and computes a result that reflects both present and past inputs. A reset port enables you to restart this tracking at any time. The running statistic is computed for each input channel independently, so the block's output is the same size as the input.

For example, in running mode (`Running` selected from the **Mode** parameter) the Maximum block outputs a record of the input's maximum value over time.

The figure below illustrates how a Maximum block in running mode operates on a frame-based 3-by-2 (two-channel) matrix input, `u`. The running maximum is reset at *t*=2 by an impulse to the block's optional `Rst` port.



Power Spectrum Estimation

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

- Burg Method (`pburg`)
- Covariance Method (`pcov`)
- Magnitude FFT (`periodogram`)
- Modified Covariance Method (`pmcov`)
- Short-Time FFT
- Yule-Walker Method (`pyulear`)

See "Spectral Analysis" in the Signal Processing Toolbox documentation for an overview of spectral analysis theory and a discussion of the above methods.

The 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 2-10 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 5-6
- "Factoring Matrices" on page 5-8
- "Inverting Matrices" on page 5-9

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

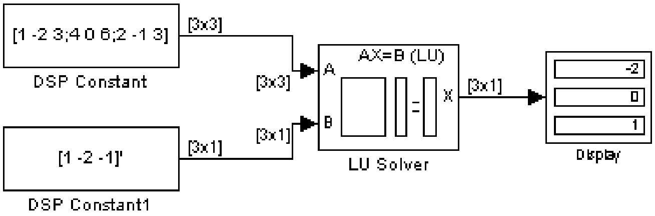
Some of the blocks offer particular strengths for certain classes of problems. For example, the Cholesky Solver block is particularly adapted for a square Hermitian positive definite matrix A, whereas the Backward Substitution block is particularly suited for an upper triangular matrix A.

Example: LU Solver

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

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

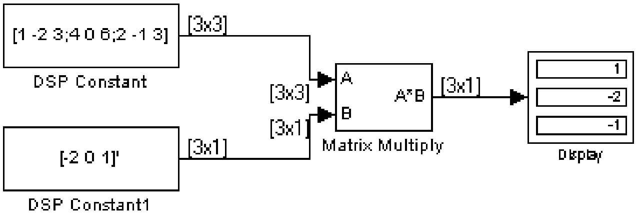
and finds x to be the vector `[-2 0 1]'`.



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:

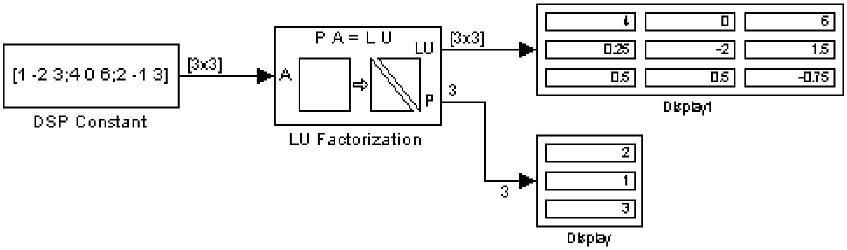
- Cholesky Factorization
- LDL Factorization
- LU Factorization
- QR Factorization
- Singular Value Decomposition

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]`.

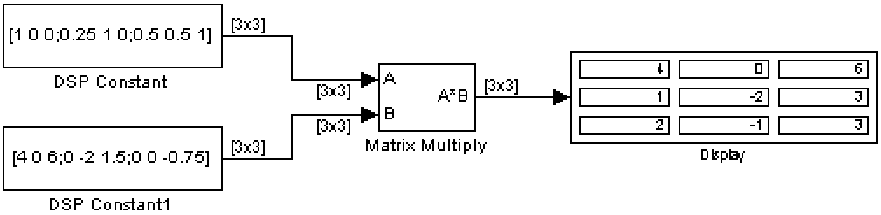
The lower output of the LU Factorization, `P`, is the permutation index vector, which indicates that the factored matrix A_p is generated from A by interchanging the first and second rows.

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

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:

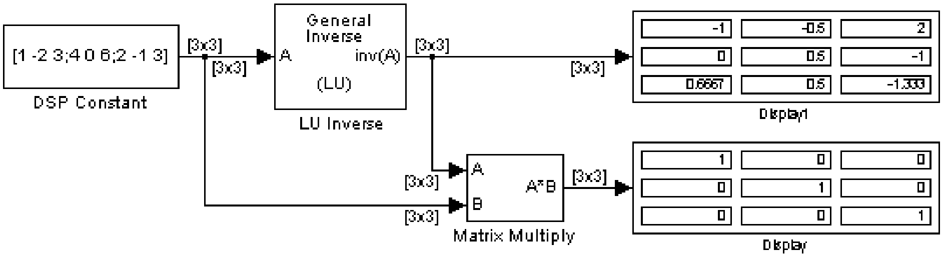
- Cholesky Inverse
- LDL Inverse
- LU Inverse
- Pseudoinverse

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

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

6

Fixed-Point Support

Fixed-Point DSP Development

Many of the blocks in the DSP Blockset have fixed-point support, so you can design DSP systems that use fixed-point arithmetic. Fixed-point support in the DSP Blockset includes

- Signed two's complement fixed-point data types
- Word lengths from 2 to 128 bits in simulation
- Word lengths from 2 to the size of a `long` on the Real-Time Workshop C code-generation target
- Overflow handling and rounding methods
- C code generation for deployment on a fixed-point embedded processor, with Real Time Workshop. The generated code uses all allowed data types supported by the embedded target, and automatically includes all necessary shift and scaling operations

Note To take full advantage of fixed-point support in the DSP Blockset, you must install the Fixed-Point Blockset. For more information, refer to the licensing information in the Fixed-Point Blockset documentation.

Benefits of Fixed-Point Hardware

There are both benefits and trade-offs to using fixed-point hardware rather than floating-point hardware for DSP development. Many DSP applications require low-power and cost-effective circuitry, which makes fixed-point hardware a natural choice. Fixed-point hardware tends to be simpler and smaller. As a result, these units require less power and cost less to produce than floating-point circuitry.

Floating-point hardware is usually larger because it demands functionality and ease of development. Floating-point hardware can accurately represent real-world numbers, and its large dynamic range reduces the risk of overflow, quantization errors, and the need for scaling. In contrast, the smaller dynamic range of fixed-point hardware that allows for low-power, inexpensive units brings the possibility of these problems. Therefore, fixed-point development must minimize the negative effects of these factors, while exploiting the benefits of fixed-point hardware; cost- and size-effective units, less power and memory usage, and fast real-time processing.

Benefits of Fixed-Point Design with the DSP Blockset

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

The DSP Blockset allows you to easily run multiple simulations with different word length, scaling, overflow handling, and rounding method choices to see the consequences of various fixed-point designs before committing to hardware. The traditional risks of fixed-point development, such as quantization errors and overflow, can be simulated and mitigated in software before going to hardware.

Fixed-point C code generation with the DSP Blockset and Real-Time Workshop produces code ready for execution on a fixed-point processor. All the choices you make in simulation with the DSP Blockset in terms of scaling, overflow handling, and rounding methods are automatically optimized in the generated code, without necessitating time-consuming and costly hand-optimized code.

Fixed-Point DSP Applications

Fixed-point support in the DSP Blockset facilitates development of a wide variety of DSP applications:

- Wireless and broadband communications
  - Cellular phones
  - Radio
  - Satellite communications
- Speech and audio processing
  - Speech processing
  - High-end audio processing

- Telephony
  - Speech coding
  - Dual tone multifrequency (DTMF)
  - Echo cancellation
- Hand-held and battery-operated consumer electronics
  - Digital recording devices
  - Personal digital assistants (PDAs)
- Computer peripherals
- Radar and sonar
- Medical electronics

Blocks with Fixed-Point Support

The following table lists all of the blocks in the DSP Blockset that support fixed-point data types in some or all modes. These blocks are colored orange in the DSP Blockset library. To take full advantage of the fixed-point capabilities of the following blocks, you must install the Fixed-Point Blockset. For more information, refer to the licensing information in the Fixed-Point Blockset documentation.

DSP Blockset Blocks with Fixed-Point Support

Autocorrelation	Buffer	Check Signal Attributes	Constant Diagonal Matrix
Convert 1-D to 2-D	Convert 2-D to 1-D	Convolution	Correlation
Counter	Create Diagonal Matrix	Data Type Conversion (Simulink block)	Delay Line
Digital Filter	Discrete Impulse	Display (Simulink block)	Downsample
DSP Constant	DSP Fixed-Point Attributes	DSP Gain	DSP Product
DSP Sum	Edge Detector	Event-Count Comparator	Extract Diagonal
Extract Triangular Matrix	FFT	Filter Realization Wizard	FIR Decimation
FIR Interpolation	Flip	Frame Status Conversion	Identity Matrix
IFFT	Integer Delay	Matrix Concatenation (Simulink block)	Matrix Product
Matrix Scaling	Matrix Sum	Matrix Viewer	Maximum
Minimum	Multiphase Clock	Multiport Selector	N-Sample Enable

DSP Blockset Blocks with Fixed-Point Support (Continued)

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	Window Function
Zero Pad			

Concepts and Terminology

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

- “Fixed-Point Data Types” on page 6-7
- “Scaling” on page 6-8
- “Precision and Range” on page 6-9

Appendix D, “Glossary of Fixed-Point Terms” defines much of the vocabulary used in these sections. For more information on these subjects, refer to the Fixed-Point Blockset documentation.

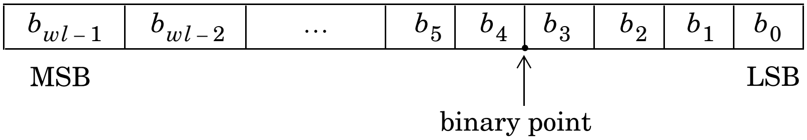
Fixed-Point Data Types

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

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

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

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



where

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

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

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

Two's complement is the most common representation of signed fixed-point numbers and is used by the DSP Blockset. Refer to "Two's Complement" on page 6-13 for more information.

Scaling

Fixed-point numbers can be encoded according to the scheme

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

where the slope can be expressed as

$$\textit{slope} = \textit{fractional slope} \times 2^{\textit{exponent}}$$

The integer is sometimes called the "*stored integer*." This is the raw binary number, in which the binary point assumed to be at the far right of the word. In the DSP Blockset, the negative of the exponent is often referred to as the "*fraction length*."

The slope and bias together represent the scaling of the fixed-point number. In a number with zero bias, only the slope affects the scaling. A fixed-point number that is only scaled by binary point position is equivalent to a number in the Fixed-Point Blockset's [Slope Bias] representation that has a bias equal to zero and a fractional slope equal to one. This is referred to as binary point-only scaling or power-of-two scaling:

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

or

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

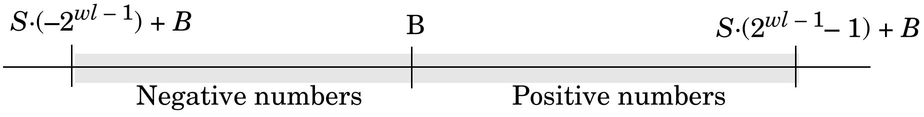
In the DSP Blockset, you can define a fixed-point data type and scaling for the output or the parameters of many blocks by specifying the word length and fraction length of the quantity. The DSP Blockset supports binary point-only scaling, so the whole of the data type and scaling information is contained in these two quantities. This is in contrast to the Fixed-Point Blockset, which supports [Slope Bias] scaling in its full generality.

Precision and Range

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

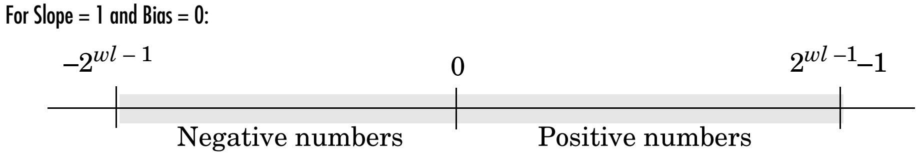
Range

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



For both the signed and unsigned fixed-point numbers of any data type, the number of different bit patterns is 2^{wl}.

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



Overflow Handling. Since a fixed-point data type represents numbers within a finite range, overflows can occur if the result of an operation is larger or smaller than the numbers in that range.

The DSP Blockset does not allow you to add guard bits to a data type on-the-fly in order to avoid overflows. Any guard bits must be allocated upon model initialization. However, the DSP Blockset does allow you to either *saturate* or *wrap* overflows. Saturation represents positive overflows as the largest positive number in the range being used, and negative overflows as the largest negative number in the range being used. Wrapping uses modulo arithmetic to cast an overflow back into the representable range of the data type. Refer to "Modulo Arithmetic" on page 6-12 for more information.

Precision

The precision of a fixed-point number is the difference between successive values representable by its data type and scaling, which is equal to the value of its least significant bit. The value of the least significant bit, and therefore the precision of the number, is determined by the number of fractional bits. A fixed-point value can be represented to within half of the precision of its data type and scaling.

For example, a fixed-point representation with four bits to the right of the binary point has a precision of 2^{-4} or 0.0625, which is the value of its least significant bit. Any number within the range of this data type and scaling can be represented to within $(2^{-4})/2$ or 0.03125, which is half the precision. This is an example of representing a number with finite precision.

Rounding Methods. One of the limitations of representing numbers with finite precision is that not every number in the available range can be represented exactly. When the result of a fixed-point calculation is a number that cannot be represented exactly by the data type and scaling being used, precision is lost. A rounding method must be used to cast the result to a representable number. The DSP Blockset currently supports `Floor` and `Nearest` rounding methods.

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

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

Arithmetic Operations

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

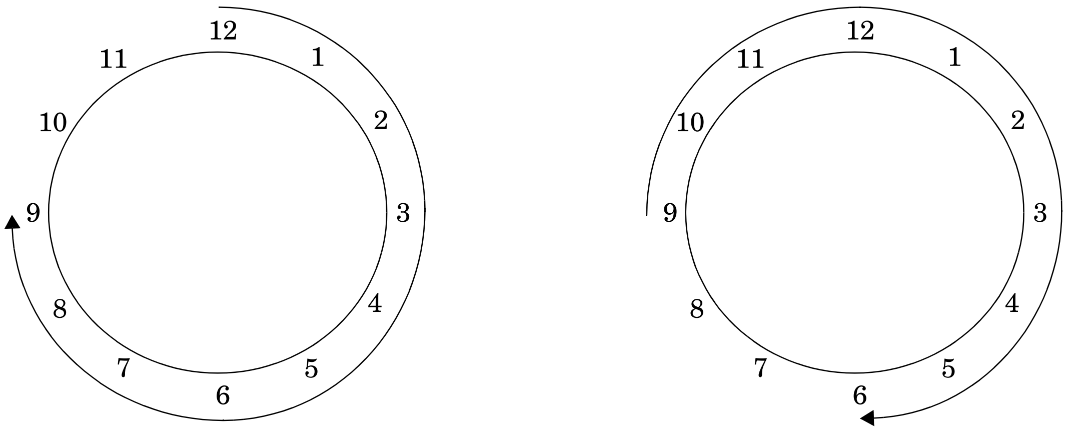
- “Modulo Arithmetic” on page 6-12
- “Two’s Complement” on page 6-13
- “Addition and Subtraction” on page 6-14
- “Multiplication” on page 6-15
- “Casts” on page 6-17

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

Modulo Arithmetic

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

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



9...

...plus 9 more...

...equals 6.

Similarly, binary math may only use the numbers 0 and 1, and any arithmetic results that fall outside of this range are wrapped “around the circle” to either 0 or 1.

Two’s Complement

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

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

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

To compute the negative of a binary number using two's complement,

1 Take the one's complement, or "flip the bits"

2 Add a "1" using binary math

3 Discard any bits carried beyond the original word length

For example, consider taking the negative of 11010 (-6). First, take the one's complement of the number, or flip the bits:

$$11010 \longrightarrow 00101$$

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

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

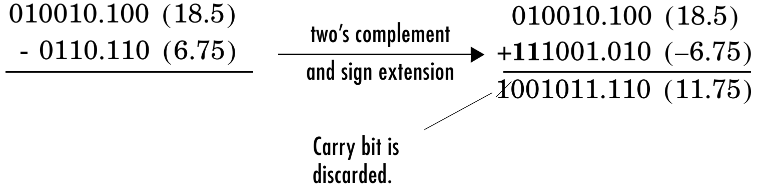
Addition and Subtraction

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

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

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

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

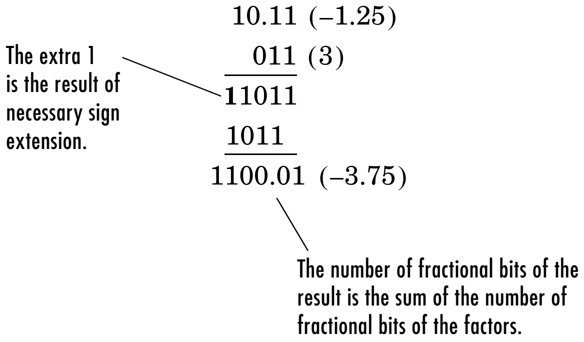


Most fixed-point DSP Blockset blocks that perform addition cast the adder inputs to an accumulator data type before performing the addition. Therefore, no further shifting is necessary during the addition to line up the binary points. Refer to “Casts” on page 6-17 for more information.

Multiplication

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

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


10.11 (–1.25)
011 (3)
11011
1011
1100.01 (–3.75)
The extra 1 is the result of necessary sign extension.
The number of fractional bits of the result is the sum of the number of fractional bits of the factors.

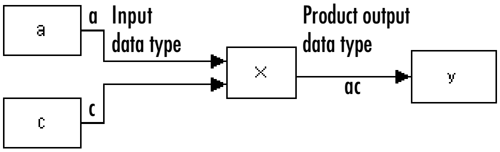
Multiplication Data Types

The following diagrams show the data types used for fixed-point multiplication in the DSP Blockset. The diagrams illustrate the differences between the data types used for real-real, complex-real, and complex-complex multiplication. Refer to individual reference pages in Chapter 7, “Block Reference” to determine whether a particular block accepts complex fixed-point inputs.

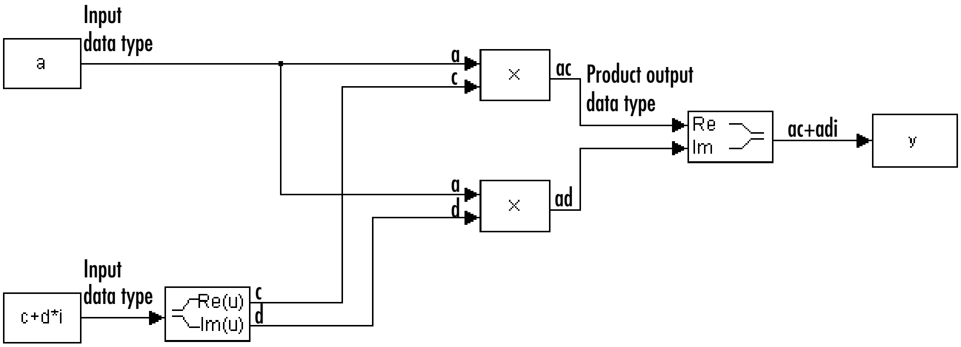
In most cases, you can set the data types used during multiplication in the block mask. Refer to “Accumulator Parameters” on page 6-24, “Product Output Parameters” on page 6-25, and “Output Parameters” on page 6-23. These data types are defined in “Casts” on page 6-17.

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

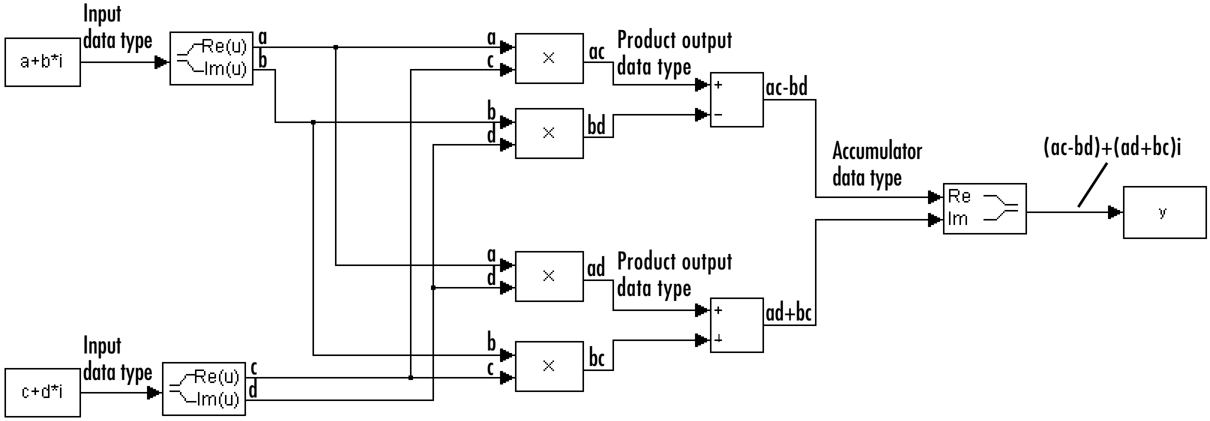
Real-Real Multiplication. The following diagram shows the data types used in the multiplication of two real numbers in the DSP Blockset. The output of this multiplication is in the product output data type:



Real-Complex Multiplication. The following diagram shows the data types used in the multiplication of a real and a complex fixed-point number in the DSP Blockset. Real-complex and complex-real multiplication are equivalent. The output of this multiplication is in the product output data type:



Complex-Complex Multiplication. The following diagram shows the multiplication of two complex fixed-point numbers in the DSP Blockset. Note that the output of the multiplication is in the accumulator data type:



Casts

Many fixed-point DSP Blockset blocks that perform arithmetic operations allow you to specify the accumulator, intermediate product, and product output data types, as applicable, as well as the output data type of the block. This section gives an overview of the casts to these data types, so that you can tell if the data types you select will invoke sign extension, padding with zeros, rounding, and/or overflow.

Casts to the Accumulator Data Type

For most fixed-point DSP Blockset blocks that perform addition, the addends are first cast to an accumulator data type. Most of the time, you may specify the accumulator data type on the block mask. Refer to “Accumulator Parameters” on page 6-24. Since the addends are both cast to the same accumulator data type before they are added together, no extra shift is necessary to insure that their binary points align. The result of the addition remains in the accumulator data type, with the possibility of overflow.

Casts to the Intermediate Product or Product Output Data Type

For DSP Blockset blocks that perform multiplication, the output of the multiplier is placed into a product output data type. Blocks that then feed the product output back into the multiplier may first cast it to an intermediate product data type. Most of the time, you may specify these data types on the block mask. Refer to “Intermediate Product Parameters” on page 6-26 and “Product Output Parameters” on page 6-25.

Casts to the Output Data Type

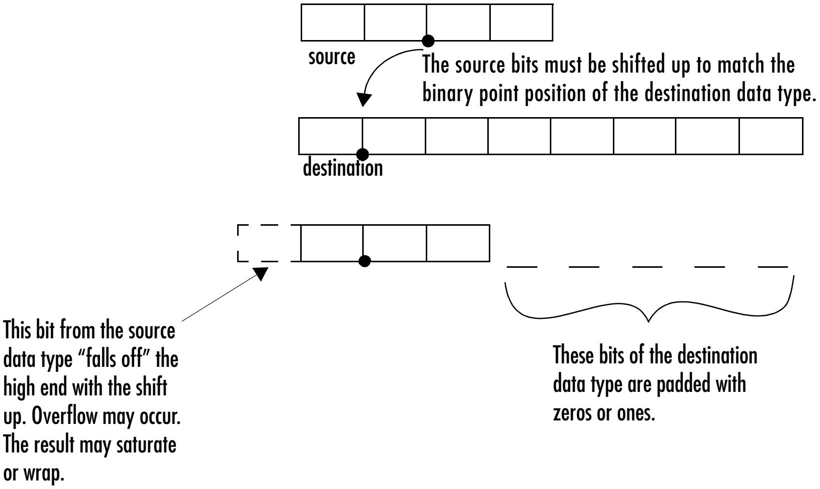
Many fixed-point DSP Blockset blocks allow you to specify the data type and scaling of the block output on the mask. Remember that the DSP Blockset does not allow mixed types on the input and output ports of its blocks. Therefore, if you would like to specify a fixed-point output data type and scaling for a DSP Blockset block that supports fixed-point data types, you must feed the input port of that block with a fixed-point signal. The final cast made by a fixed-point DSP Blockset block is to the output data type of the block.

Note that although you may not mix fixed-point and floating-point signals on the input and output ports of DSP Blockset blocks, you may have fixed-point signals with different word and fraction lengths on the ports of blocks that support fixed-point signals.

Casting Examples

It is important to keep in mind the ramifications of each cast when selecting these intermediate data types, as well as any other intermediate fixed-point data types that may be allowed by a particular block. Depending upon the data types you select, overflow and/or rounding may occur. The following two examples demonstrate cases where overflow and rounding can occur.

Casting from a Shorter Data Type to a Longer Data Type. Consider the cast of a nonzero number represented by a four-bit data type with two fractional bits, to an eight-bit data type with seven fractional bits:

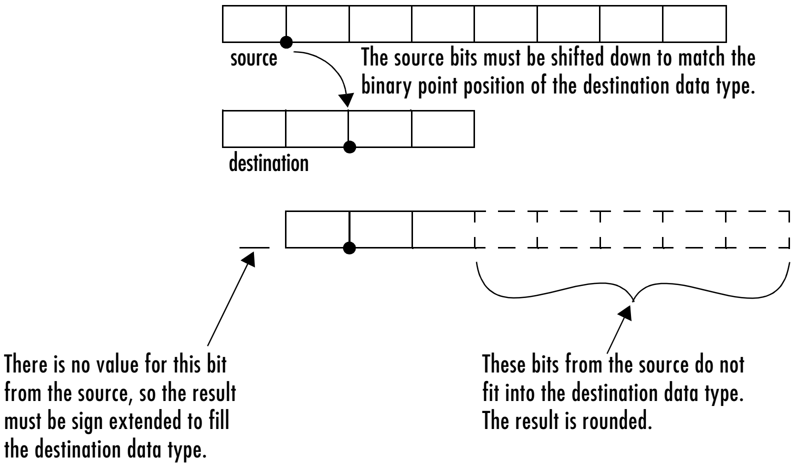

source
The source bits must be shifted up to match the binary point position of the destination data type.
destination
This bit from the source data type "falls off" the high end with the shift up. Overflow may occur. The result may saturate or wrap.
These bits of the destination data type are padded with zeros or ones.

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

- If overflow does not occur, the empty bits are padded with zeros.
- If wrapping occurs, the empty bits are padded with zeros.
- If saturation occurs
  - The empty bits of a positive number are padded with ones.
  - The empty bits of a negative number are padded with zeros.

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

Casting from a Longer Data Type to a Shorter Data Type. Consider the cast of a nonzero number represented by an eight-bit data type with seven fractional bits, to a four-bit data type with two fractional bits:


source
The source bits must be shifted down to match the binary point position of the destination data type.
destination
There is no value for this bit from the source, so the result must be sign extended to fill the destination data type.
These bits from the source do not fit into the destination data type. The result is rounded.

As the diagram shows, the source bits are shifted down so that the binary point matches the destination binary point position. There is no value for the highest bit from the source, so the result is sign extended to fill the integer portion of the destination data type. The bottom five bits of the source do not fit into the fraction length of the destination. Therefore, precision may be lost as the result is rounded.

In this case, even though the cast was from a longer data type to a shorter data type, all the integer bits were maintained. Conversely, full precision can be maintained even if you cast to a shorter data type, as long as the fraction length of the destination data type is the same length or longer than the fraction length of the source data type. In that case, however, bits will be lost from the high end of the result and overflow may occur.

The worst-case scenario occurs when both the integer length and the fraction length of the destination data type are shorter than those of the source data type and scaling. In that case, both overflow and a loss of precision can occur.

Specifying Fixed-Point Attributes

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

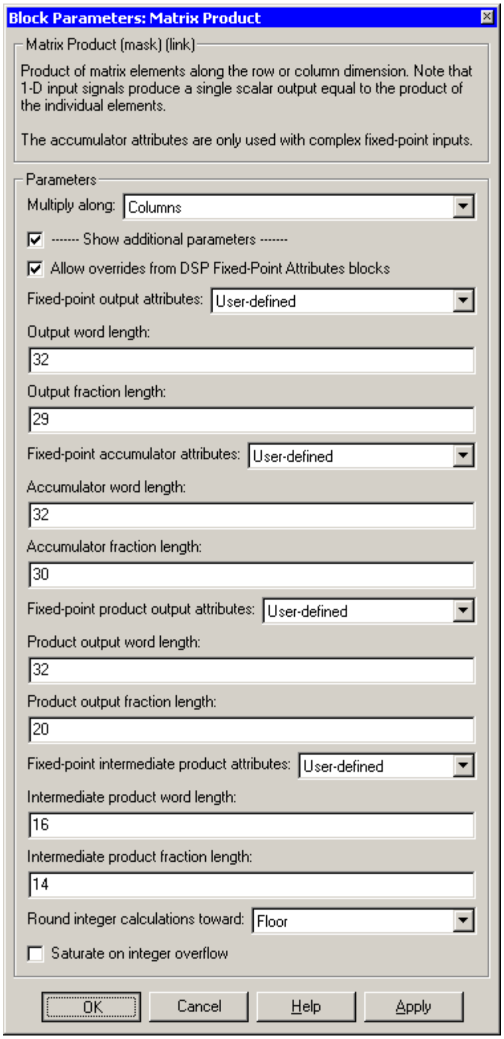
- “Setting Block Parameters” on page 6-21
- “Specifying System-Level Settings” on page 6-27

Setting Block Parameters

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

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

Most fixed-point parameters for DSP Blockset blocks appear when the **Show additional parameters** check box is selected, for example on the Matrix Product and FIR Decimation blocks below.



Block Parameters: Matrix Product

Matrix Product (mask) (link)

Product of matrix elements along the row or column dimension. Note that 1-D input signals produce a single scalar output equal to the product of the individual elements.

The accumulator attributes are only used with complex fixed-point inputs.

Parameters

Multiply along: Columns

☑ ------- Show additional parameters -------

☑ Allow overrides from DSP Fixed-Point Attributes blocks

Fixed-point output attributes: User-defined

Output word length: 32

Output fraction length: 29

Fixed-point accumulator attributes: User-defined

Accumulator word length: 32

Accumulator fraction length: 30

Fixed-point product output attributes: User-defined

Product output word length: 32

Product output fraction length: 20

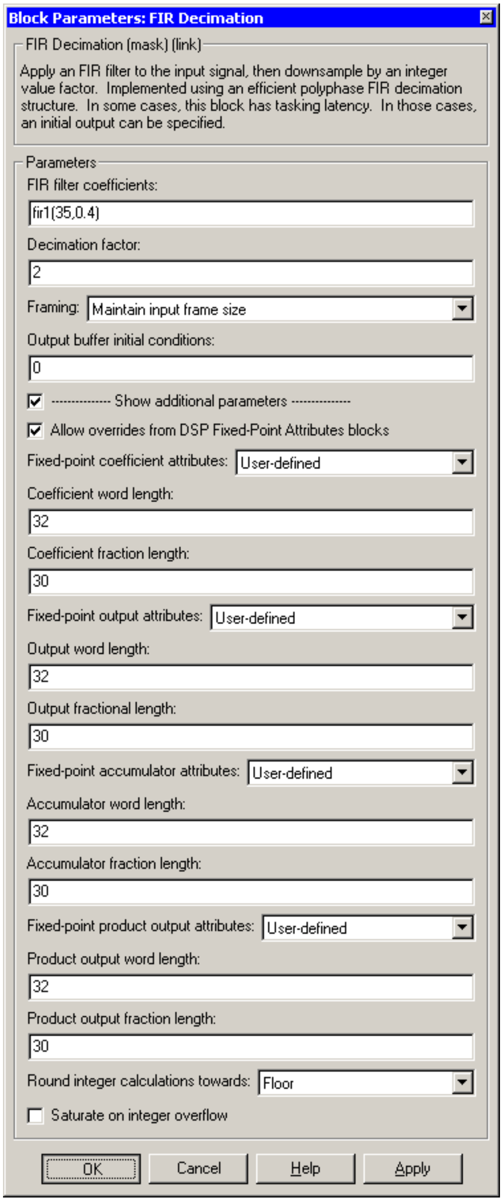
Fixed-point intermediate product attributes: User-defined

Intermediate product word length: 16

Intermediate product fraction length: 14

Round integer calculations toward: Floor

□ Saturate on integer overflow



Block Parameters: FIR Decimation

FIR Decimation (mask) (link)

Apply an FIR filter to the input signal, then downsample by an integer value factor. Implemented using an efficient polyphase FIR decimation structure. In some cases, this block has tasking latency. In those cases, an initial output can be specified.

Parameters

FIR filter coefficients: fir1(35,0.4)

Decimation factor: 2

Framing: Maintain input frame size

Output buffer initial conditions: 0

☑ -------------- Show additional parameters --------------

☑ Allow overrides from DSP Fixed-Point Attributes blocks

Fixed-point coefficient attributes: User-defined

Coefficient word length: 32

Coefficient fraction length: 30

Fixed-point output attributes: User-defined

Output word length: 32

Output fractional length: 30

Fixed-point accumulator attributes: User-defined

Accumulator word length: 32

Accumulator fraction length: 30

Fixed-point product output attributes: User-defined

Product output word length: 32

Product output fraction length: 30

Round integer calculations towards: Floor

□ Saturate on integer overflow

Many of the DSP Blockset blocks with fixed-point capabilities share common parameters, though each block may have a different subset of these fixed-point parameters. The following parameters are discussed in this section:

- "Allow Overrides from DSP Fixed-Point Attributes Blocks Parameter" on page 6-23
- "Output Parameters" on page 6-23
- "Accumulator Parameters" on page 6-24
- "Product Output Parameters" on page 6-25
- "Intermediate Product Parameters" on page 6-26
- "Round Integer Calculations Toward Parameter" on page 6-26
- "Saturate on Integer Overflow Parameter" on page 6-27

For a discussion of all the parameters of a specific DSP Blockset block, refer to the block's reference page in Chapter 7, "Block Reference."

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

Allow Overrides from DSP Fixed-Point Attributes Blocks Parameter

This parameter, which is selected by default, allows fixed-point parameters to be set at the system or subsystem level. Refer to "DSP Fixed-Point Attributes Block" on page 6-28.

Output Parameters

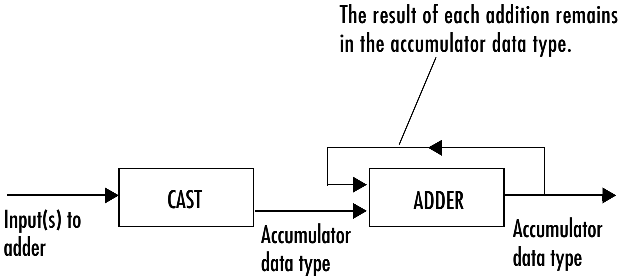
In many cases you may specify the output data type and scaling of fixed-point DSP Blockset blocks:

- The **Fixed-point output attributes** parameter allows you to specify whether you want the output data type and scaling to automatically be the same as the input, or whether you'd like to specify the output word length and fraction length yourself in the dialog.
- The **Output word length** parameter allows you to specify the word length of the output, if you chose to do so in the **Fixed-point output attributes** parameter.

- The **Output fraction length** parameter allows you to specify the fraction length of the output, if you chose to do so in the **Fixed-point output attributes** parameter.

Accumulator Parameters

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

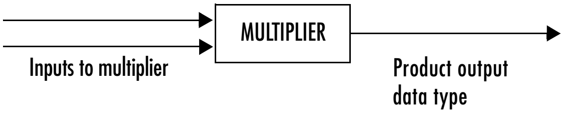


Refer to the reference page of a specific block in Chapter 7, "Block Reference" for details on the accumulator data type of a specific block:

- The **Fixed-point accumulator attributes** parameter allows you to specify whether you want the accumulator data type and scaling to automatically be the same as the output or an input, or whether you'd like to specify the accumulator word length and fraction length yourself in the dialog.
- The **Accumulator word length** parameter allows you to specify the word length of the accumulator, if you chose to do so in the **Fixed-point accumulator attributes** parameter.
- The **Accumulator fraction length** parameter allows you to specify the fraction length of the accumulator, if you chose to do so in the **Fixed-point accumulator attributes** parameter.

Product Output Parameters

Fixed-point DSP Blockset blocks that must hold multiplication results for further calculation usually allow you to specify the data type and scaling of the product output:

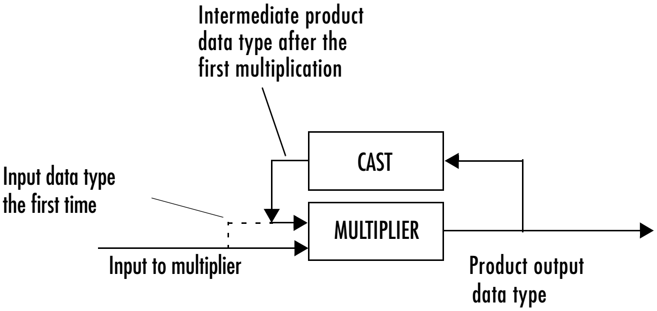


Refer to the reference page of a specific block in Chapter 7, “Block Reference” to learn about the product output data type for a specific block. Note that for complex-complex multiplication, the multiplication result is in the accumulator data type. Refer to “Multiplication Data Types” on page 6-15 for more information on complex fixed-point multiplication in the DSP Blockset:

- The **Fixed-point product output attributes** parameter allows you to specify whether you want the product output data type and scaling to automatically be the same as the output, an input, or the accumulator; or whether you’d like to specify the product output word length and fraction length yourself in the dialog.
- The **Product output word length** parameter allows you to specify the word length of the product output, if you chose to do so in the **Fixed-point product output attributes** parameter.
- The **Product output fraction length** parameter allows you to specify the fraction length of the product output, if you chose to do so in the **Fixed-point product output attributes** parameter.

Intermediate Product Parameters

Fixed-point DSP Blockset blocks that feed multiplication results back to the input of the multiplier usually allow you to specify the data type and scaling of the intermediate product:


Intermediate product data type after the first multiplication
CAST
Input data type the first time
MULTIPLIER
Input to multiplier
Product output data type

Refer to the reference page of a specific block in Chapter 7, “Block Reference” to learn about the intermediate product data type for a specific block:

- The **Fixed-point intermediate product attributes** parameter allows you to specify whether you want the intermediate product data type and scaling to automatically be the same as the input, or whether you’d like to specify the intermediate product word length and fraction length yourself in the dialog.
- The **Intermediate product word length** parameter allows you to specify the word length of the intermediate product, if you chose to do so in the **Fixed-point intermediate product attributes** parameter.
- The **Intermediate product fraction length** parameter allows you to specify the fraction length of the intermediate product, if you chose to do so in the **Fixed-point intermediate product attributes** parameter.

Round Integer Calculations Toward Parameter

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

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

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

Saturate on Integer Overflow Parameter

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

- If you select this parameter, positive overflows will saturate to the largest positive number of the data type. Negative overflows will saturate to the largest negative number.
- If you do not select this parameter, overflows will wrap.

Specifying System-Level Settings

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

Fixed-Point Settings Interface

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

- The `fxptdlg` reference page — A reference page on the Fixed-Point Settings interface
- Chapter 6, “Tutorial: Feedback Controller Simulation” — A tutorial that highlights the use of the Fixed-Point Settings interface

Logging. The Fixed-Point Settings interface logs overflows and saturations for fixed-point DSP Blockset blocks. The Fixed-Point Settings interface does not log overflows and saturations when the `Data overflow` line in the **Diagnostics** tab of the **Simulation Parameters** dialog is set to `None`.

The Fixed-Point Settings interface does not log the simulation minimums and maximums for fixed-point DSP Blockset blocks. Therefore, the autoscaling tool cannot be used for these blocks.

Data type override. DSP Blockset blocks obey the `Use local settings`, `True doubles`, `True singles`, and `Force off` modes of the **Data type override** parameter in the Fixed-Point Settings interface. The `Scaled doubles` mode is also supported for DSP Blockset source and byte-shuffling blocks that support [Slope Bias] signals, but not for arithmetic fixed-point DSP Blockset blocks such as FFT or Digital Filter.

DSP Fixed-Point Attributes Block

For large models, configuring the parameters of each DSP Blockset block with fixed-point support can be very time consuming. The DSP Fixed-Point Attributes block mitigates this problem by enabling you to set the following parameters on the system or subsystem level.

On nonsource blocks:

- **Output word length**
- **Output fraction length**
- **Accumulator word length**
- **Accumulator fraction length**
- **Product output word length**
- **Product output fraction length**
- **State memory word length**
- **State memory fraction length**
- **Round integer calculations toward**
- **Saturate on integer overflow**

On source blocks:

- **Word length**
- **Set fraction length in output to**
- **Fraction length**

You can set all these fixed-point parameters for all applicable blocks in this single high-level GUI. For more information on this block, refer to its reference page in Chapter 7, “Block Reference.”

Fixed-Point Filtering

The following DSP Blockset blocks enable you to design and/or realize a variety of fixed-point filters:

- Digital Filter
- Filter Realization Wizard
- FIR Decimation
- FIR Interpolation

The FIR Decimation, FIR Interpolation, and Digital Filter blocks are implementation blocks. They allow you to implement filters for which you already know the filter coefficients. The FIR Decimation block and the FIR Interpolation block each implement their respective filter type, while the Digital Filter can create a variety of filter structures. The Digital Filter block currently supports FIR Direct Form and IIR Biquadratic Direct Form II Transposed (SOS) filters for fixed-point signals. For more information on these blocks, refer to their reference pages in Chapter 7, “Block Reference.”

The Filter Realization Wizard block invokes part of the Filter Design and Analysis Tool from the Signal Processing Toolbox. This block allows you both to design new filters and to implement filters for which you already know the coefficients. In its implementation stage, the Filter Realization Wizard creates a filter realization using Sum, Gain, and Integer Delay blocks. You can use this block to design and/or implement numerous types of fixed-point and floating-point filters. Refer to Chapter 3, “Filters” and the Filter Realization Wizard reference page in Chapter 7, “Block Reference” for more information.

Interoperability with Other Products

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

- "Fixed-Point Data Type Support" on page 6-30
- "Fixed-Point Scaling Support" on page 6-32
- "Fixed-Point Operations Support" on page 6-33
- "Fixed-Point Code Generation Support" on page 6-33

Fixed-Point Data Type Support

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

Fixed-Point Data Type Support (Continued)

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

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

[2] Fixed-point DSP Blockset blocks that only manipulate bits and do not perform arithmetic operations accept unsigned integer, fractional, and generalized fixed-point inputs. The following source blocks also allow you to specify these types of outputs: Constant Diagonal Matrix, Discrete Impulse, DSP Constant, and Identity Matrix.

[3] DSP Blockset blocks obey the `Use local settings`, `True doubles`, `True singles`, and `Force off` modes of the **Data type override** parameter in the **Fixed-Point Settings** interface. The `Scaled doubles` mode is also supported for DSP Blockset source and byte-shuffling blocks that support [Slope Bias] signals, but not for other arithmetic fixed-point DSP Blockset blocks.

Fixed-Point Scaling Support

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

[1] Fixed-point DSP Blockset blocks that only manipulate bits and do not perform arithmetic operations accept [Slope Bias] signals. The following source blocks also allow you to specify a [Slope Bias] output signal: Constant Diagonal Matrix, Discrete Impulse, DSP Constant, Identity Matrix, and Sine Wave.

Fixed-Point Operations Support

	DSP Blockset Fixed-Point	Filter Design Toolbox	Simulink Fixed-Point Blocks/ Fixed-Point Blockset Blocks	Stateflow
Rounding methods	Floor, nearest	Ceiling, convergent, fix, floor, round	Ceiling, floor, nearest, zero	Offline conversions are rounded to nearest Online conversions are rounded to floor or zero
Overflow handling	Saturate, wrap	Saturate, wrap	Saturate, wrap	Simulation halts upon overflow
Logging	Partial support via the Fixed-Point Settings interface[1]	Yes, via the `qreport` function	Yes, via the Fixed-Point Settings interface	No

[1] Simulation overflows and saturations are logged for DSP Blockset blocks in the **Fixed-Point Settings** interface. Simulation minimums and maximums are not logged for these blocks.

Fixed-Point Code Generation Support

	DSP Blockset Fixed-Point	Filter Design Toolbox	Simulink Fixed-Point Blocks/ Fixed-Point Blockset Blocks	Stateflow
C code generation	Yes	No	Yes	Yes

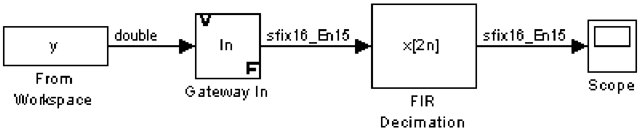
Building Models with Other Blocks

You can build models with fixed-point DSP Blockset blocks that include fixed-point and floating-point blocks both from the DSP Blockset and from other MathWorks products. The following sections discuss issues to keep in mind when connecting fixed-point DSP Blockset blocks to other types of blocks.

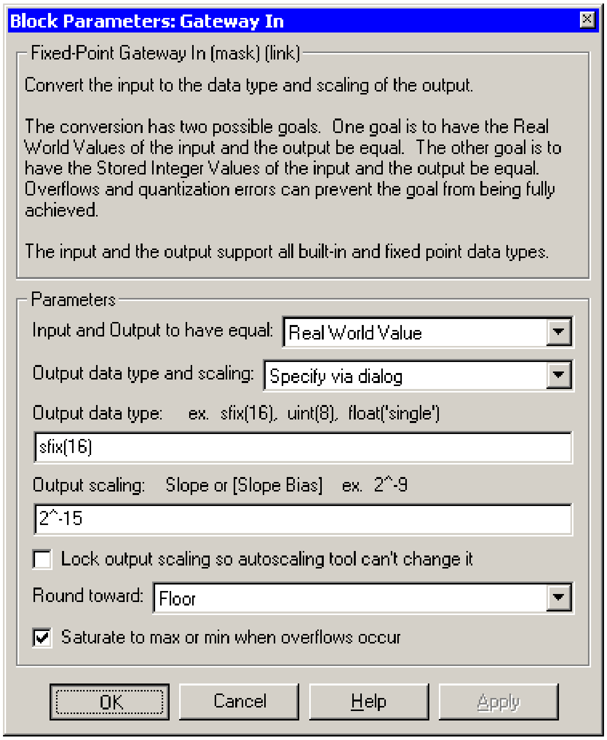
Connecting Fixed-Point and Floating-Point Blocks

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

To feed a DSP Blockset block with a fixed-point signal from another block that does not have fixed-point support, use the Gateway In block from the Fixed-Point Blockset, as in the model below:



The Simulink From Workspace block in the model does not allow you to set a fixed-point output data type and scaling in its block mask. The Gateway In block, however, allows you to do so. The following shows the mask parameter settings of the Gateway In block in the model:



Block Parameters: Gateway In

Fixed-Point Gateway In (mask) (link)

Convert the input to the data type and scaling of the output.

The conversion has two possible goals. One goal is to have the Real World Values of the input and the output be equal. The other goal is to have the Stored Integer Values of the input and the output be equal. Overflows and quantization errors can prevent the goal from being fully achieved.

The input and the output support all built-in and fixed point data types.

Parameters

Input and Output to have equal: Real World Value

Output data type and scaling: Specify via dialog

Output data type: ex. sfix(16), uint(8), float('single')

sfix(16)

Output scaling: Slope or [Slope Bias] ex. 2^-9

2^-15

□ Lock output scaling so autoscaling tool can't change it

Round toward: Floor

☑ Saturate to max or min when overflows occur

Note that the **Output scaling** parameter of the Gateway In block specifies a power-of-two scaling with 0 bias. This is a requirement for fixed-point signals in the DSP Blockset, as discussed in the following section.

Similarly to the example above, you can feed the output of fixed-point DSP Blockset blocks to other blocks that do not accept fixed-point data types by using the Fixed-Point Blockset Gateway Out block.

Connecting Blocks with Different Scalings

Fixed-point signals in the DSP Blockset must have a fractional slope of 1 and a bias of 0; that is, only power-of-two or binary point-only scaling is accepted. You must make sure that any block that feeds the input port of a fixed-point DSP Blockset block specifies binary point-only scaling for the output scaling of the block. Alternately, you can use the Fixed-Point Blockset Conversion block between any two fixed-point blocks of with different scalings.

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 find out about using blocks together for common DSP tasks, see Chapter 2, "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 7-21.)

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.

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.

Chirp	Generate a swept-frequency cosine (chirp) signal
Constant Diagonal Matrix	Generate a square, diagonal matrix
Discrete Impulse	Generate a discrete impulse

DSP Constant	Generate a discrete-time or continuous-time constant signal
DSP Fixed-Point Attributes	Set fixed-point attributes of DSP Blockset blocks on the system or subsystem level
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

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.

Autocorrelation LPC	Determine the coefficients of an Nth-order forward linear predictor
Levinson-Durbin	Solve a linear system of equations using Levinson-Durbin recursion

LPC to LSF/LSP Conversion	Convert linear prediction coefficients (LPCs) to line spectral pairs (LSPs) or line spectral frequencies (LSFs)
LPC to/from RC	Convert linear prediction coefficients (LPCs) to reflection coefficients (RCs) or reflection coefficients to linear prediction coefficients
LPC/RC to Autocorrelation	Convert linear prediction coefficients (LPCs) or reflection coefficients (RCs) to autocorrelation coefficients (ACs)
LSF/LSP to LPC Conversion	Convert line spectral 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.

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.

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 Designs"
- "Multirate Filters"

Adaptive Filters

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

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

Filter Designs

Blocks for designing and implementing various filters.

Analog Filter Design	Design and implement an analog filter
Digital Filter Design	Design, analyze, and implement a variety of digital FIR and IIR filters
Digital Filter	Filter inputs with a specified time-varying or static digital FIR or IIR filter
DSP Fixed-Point Attributes	Set fixed-point attributes of DSP Blockset blocks on the system or subsystem level
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.

DSP Fixed-Point Attributes	Set fixed-point attributes of DSP Blockset blocks on the system or subsystem level
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.

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

DSP Fixed-Point Attributes	Set fixed-point attributes of DSP Blockset blocks on the system or subsystem level
DSP Gain	Multiply the input by a constant
DSP Product	Perform element-wise multiplication of two inputs
DSP Sum	Add two inputs
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.

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.

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.

LU Inverse	Compute the inverse of a square matrix using LU factorization
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
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.

Constant Diagonal Matrix	Generate a square, diagonal matrix
Create Diagonal Matrix	Create a square diagonal matrix from diagonal elements
DSP Fixed-Point Attributes	Set fixed-point attributes of DSP Blockset blocks on the system or subsystem level
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 Exponential	Compute the matrix exponential
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

Matrix 1-Norm	Compute the 1-norm of a matrix
Matrix Multiply	Multiply input matrices
Overwrite Values	Overwrite a submatrix or subdiagonal of the input
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.

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.

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.

Quantizer (Simulink block)	Discretize input at a specified interval
Scalar Quantizer	Convert an input signal into a set of quantized output values. Convert an input signal into a set of index values. Convert a set of index values into a quantized output signal.
Scalar Quantizer Design	Start the Scalar Quantizer Design Tool (SQDTool) to design a scalar quantizer using the Lloyd algorithm
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.

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.

Flip	Flip the input vertically or horizontally
Multiport Selector	Distribute arbitrary subsets of input rows or columns to multiple output ports
Overwrite Values	Overwrite a submatrix or subdiagonal of the input

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.

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
DSP Fixed-Point Attributes	Set fixed-point attributes of DSP Blockset blocks on the system or subsystem level
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.

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 operations on a signal.

Constant Ramp	Generate a ramp signal with length based on input dimensions
Convolution	Compute the convolution of two inputs
Delay	Delay the discrete-time input by a specified number of samples or frames
Downsample	Resample an input at a lower rate by deleting samples
DSP Fixed-Point Attributes	Set fixed-point attributes of DSP Blockset blocks on the system or subsystem level
Interpolation	Interpolate values of real input samples
Pad	Alter the input size by padding or truncating rows and/or columns
Repeat	Resample an input at a higher rate by repeating values
Sample and Hold	Sample and hold an input signal
Triggered Signal From Workspace	Import signal samples from the MATLAB workspace when triggered
Unwrap	Unwrap the phase of a signal
Upsample	Resample an input at a higher rate by inserting zeros
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

Window Function	Compute a window, and/or apply a window to an input signal
Zero Pad	Alter the input size by zero-padding or truncating rows and/or columns

Statistics

Blocks for performing various statistical computations.

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
DSP Fixed-Point Attributes	Set fixed-point attributes of DSP Blockset blocks on the system or subsystem level
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.

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
DSP Fixed-Point Attributes	Set fixed-point attributes of DSP Blockset blocks on the system or subsystem level
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

Analog Filter Design

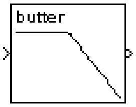
Purpose

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.

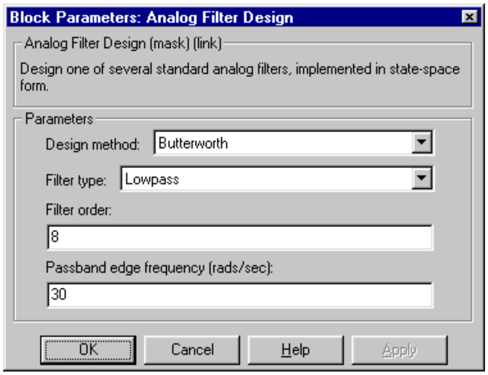
Filter Design	Description
`Butterworth`	The magnitude response of a Butterworth filter is maximally flat in the passband and monotonic overall.
`Chebyshev type I`	The magnitude response of a Chebyshev type I filter is equiripple in the passband and monotonic in the stopband.
`Chebyshev type II`	The magnitude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



Block Parameters: Analog Filter Design

Analog Filter Design (mask) (link)

Design one of several standard analog filters, implemented in state-space form.

Parameters

Design method: Butterworth

Filter type: Lowpass

Filter order:

8

Passband edge frequency (rads/sec):

30

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

Design method

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

Filter type

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

Filter order

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

Passband edge frequency

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

Lower passband edge frequency

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

Upper passband edge frequency

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

Stopband edge frequency

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

Lower stopband edge frequency

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

Upper stopband edge frequency

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

Passband ripple in dB

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

Stopband attenuation in dB

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

References

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

Supported Data Types

- Double-precision floating point

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

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:

- Chapter 3, "Filters"
- "Analog Filter Design Block" on page 3-44

Analytic Signal

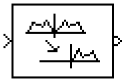
Purpose

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

$$y = u + j\mathrm{H}\{u\}$$

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

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

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

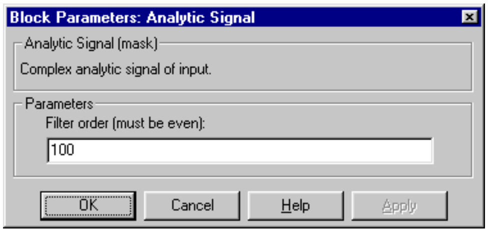
Sample-Based Operation

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

Frame-Based Operation

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

Dialog Box



Block Parameters: Analytic Signal

Analytic Signal (mask)

Complex analytic signal of input.

Parameters

Filter order (must be even): 100

Filter order

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

Supported Data Types

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

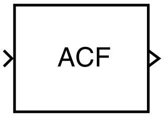
To learn how to convert your data types to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-2.

Autocorrelation

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 non-negative lag (less than input length)** parameter.

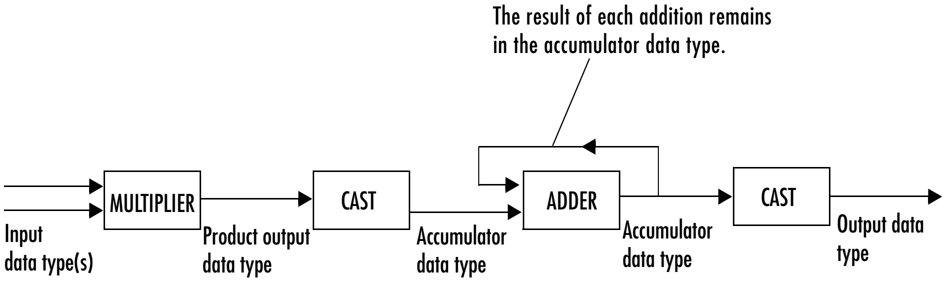
Input u is zero when indexed outside of its valid range. When the input is real, the output is real; otherwise, the output is complex.

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 Autocorrelation block accepts both real and complex floating-point inputs. It also accepts real fixed-point inputs.

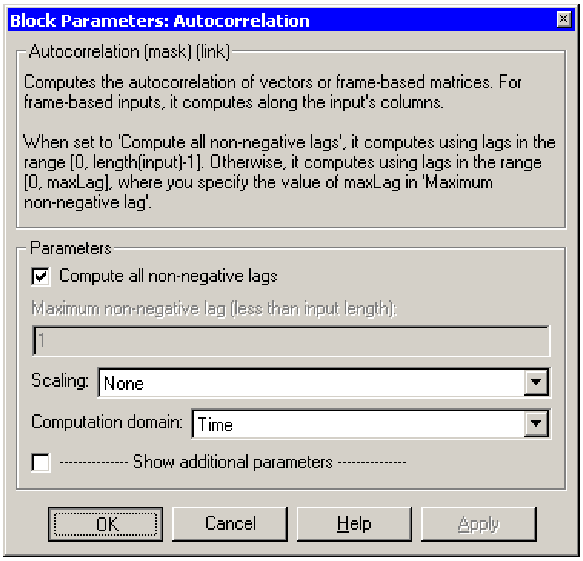
Fixed-Point Data Types

The following diagram shows the data types used within the Autocorrelation block for fixed-point signals.



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

Dialog Box



Block Parameters: Autocorrelation

Autocorrelation (mask) (link)

Computes the autocorrelation of vectors or frame-based matrices. For frame-based inputs, it computes along the input's columns.

When set to 'Compute all non-negative lags', it computes using lags in the range [0, length(input)-1]. Otherwise, it computes using lags in the range [0, maxLag], where you specify the value of maxLag in 'Maximum non-negative lag'.

Parameters

☑ Compute all non-negative lags

Maximum non-negative lag (less than input length): 1

Scaling: None

Computation domain: Time

☐ -------------- Show additional parameters --------------

Compute all non-negative lags

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

Maximum non-negative lag (less than input length)

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

Scaling

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

- `None` — Generates the raw autocorrelation, $y_{i,j}$, without normalization
- `Biased` — Generates the biased estimate of the autocorrelation

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

- `Unbiased` — Generates the unbiased estimate of the autocorrelation

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

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

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

The **Scaling** parameter must be set to `None` for fixed-point signals. Tunable, except in the Simulink external mode.

Computation domain

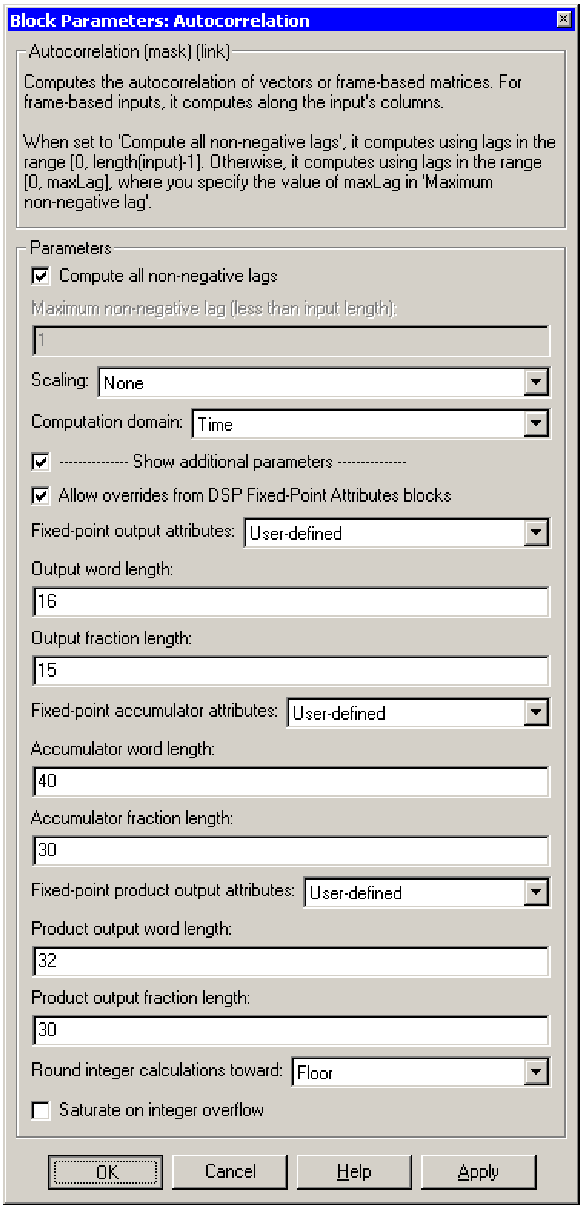
This parameter sets the domain in which the block computes convolutions to one of the following settings:

- `Time` — 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

This parameter must be set to `Time` for fixed-point signals.

Show additional parameters

If selected, additional parameters specific to implementation of the block become visible as shown.



Block Parameters: Autocorrelation

Autocorrelation (mask) (link)

Computes the autocorrelation of vectors or frame-based matrices. For frame-based inputs, it computes along the input's columns.

When set to 'Compute all non-negative lags', it computes using lags in the range [0, length(input)-1]. Otherwise, it computes using lags in the range [0, maxLag], where you specify the value of maxLag in 'Maximum non-negative lag'.

Parameters

☑ Compute all non-negative lags

Maximum non-negative lag (less than input length): 1

Scaling: None

Computation domain: Time

☑ ------------- Show additional parameters -------------

☑ Allow overrides from DSP Fixed-Point Attributes blocks

Fixed-point output attributes: User-defined

Output word length: 16

Output fraction length: 15

Fixed-point accumulator attributes: User-defined

Accumulator word length: 40

Accumulator fraction length: 30

Fixed-point product output attributes: User-defined

Product output word length: 32

Product output fraction length: 30

Round integer calculations toward: Floor

□ Saturate on integer overflow

Allow overrides from DSP Fixed-Point Attributes blocks

If you select this parameter, fixed-point data types for this block may be set by DSP Fixed-Point Attributes blocks in your model. If this parameter is unselected, the data types are always set by the parameters in the block mask.

Fixed-point output attributes

Choose how you will specify the output word length and fraction length. If you select `Same as input`, these characteristics will match those of the input to the block. If you select `User-defined`, the **Output word length** and **Output fraction length** parameters become visible.

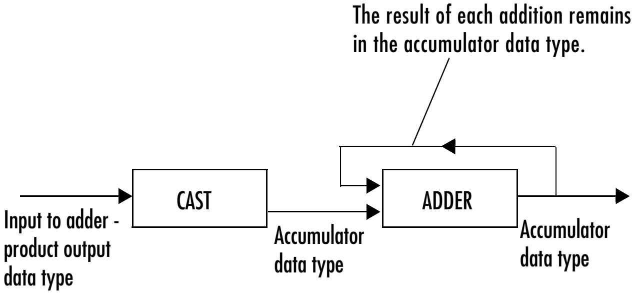
Output word length

Specify the word length, in bits, of the output. This parameter is only visible if `User-defined` is specified for the **Fixed-point output attributes** parameter.

Output fraction length

Specify the fraction length, in bits, of the output. This parameter is only visible if `User-defined` is specified for the **Fixed-point output attributes** parameter.

Fixed-point accumulator attributes


The result of each addition remains in the accumulator data type.
CAST
ADDER
Input to adder - product output data type
Accumulator data type
Accumulator data type

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

If you select `Same as output`, the accumulator word and fraction lengths are the same as those of the output of the block. If you select `User-defined`, the **Accumulator word length** and **Accumulator fraction length** parameters become visible.

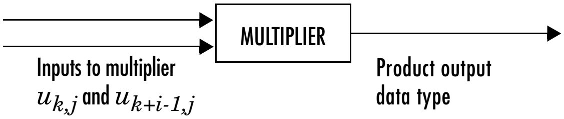
Accumulator word length

Specify the word length, in bits, of the accumulator. This parameter is only visible when `User-defined` is specified for the **Fixed-point accumulator attributes** parameter.

Accumulator fraction length

Specify the fraction length, in bits, of the accumulator. This parameter is only visible when `User-defined` is specified for the **Fixed-point accumulator attributes** parameter.

Fixed-point product output attributes



As depicted above, the output of the multiplier is placed into the product output data type and scaling. Use this parameter to specify how you would like to designate this product output word and fraction lengths.

If you select `Same as accumulator`, the product output word and fraction lengths are the same as those of the accumulator of the block. If you select `Same as output`, they are the same as those of the output of the block. If you select `User-defined`, the **Product output word length** and **Product output fraction length** parameters become visible.

Product output word length

Specify the word length, in bits, of the product output. This parameter is only visible when `User-defined` is specified for the **Fixed-point product output attributes** parameter.

Product output fraction length

Specify the fraction length, in bits, of the product output. This parameter is only visible when `User-defined` is specified for the **Fixed-point product output attributes** parameter.

Round integer calculations toward

Select the rounding mode for fixed-point operations.

Saturate on integer overflow

If selected, overflows saturate.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point
- 8-, 16-, and 32-bit signed integers

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

See Also

Correlation	DSP Blockset
`xcorr`	Signal Processing Toolbox

Autocorrelation LPC

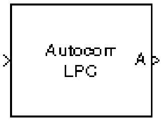
Purpose

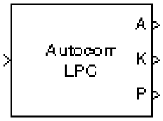
Determine the coefficients of an Nth-order forward linear predictor

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.

$$\hat{u}_{M+1} = -(a_2u_M)-(a_3u_{M-1})-\cdots-(a_{N+1}u_{M-N+1})$$

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

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

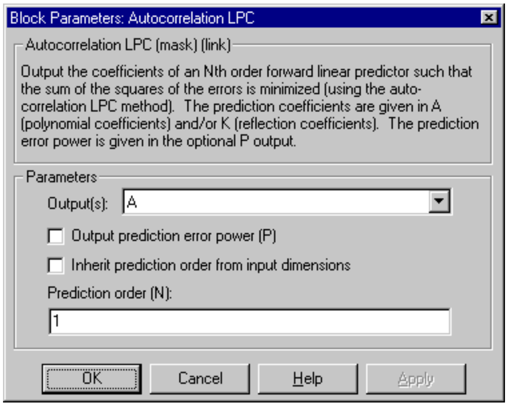
leads to the system of equations

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

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



Block Parameters: Autocorrelation LPC

Autocorrelation LPC (mask) (link)

Output the coefficients of an Nth order forward linear predictor such that the sum of the squares of the errors is minimized (using the autocorrelation LPC method). The prediction coefficients are given in A (polynomial coefficients) and/or K (reflection coefficients). The prediction error power is given in the optional P output.

Parameters

Output(s): A

□ Output prediction error power (P)

□ Inherit prediction order from input dimensions

Prediction order (N): 1

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 your data types to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-2.

See Also

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

Backward Substitution

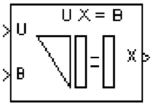
Purpose

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

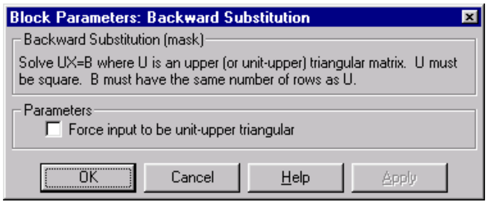


The Backward Substitution block solves the linear system U*X*=B by simple backward substitution of variables, where U is the upper triangular M-by-M matrix input to the `U` port, and B is the M-by-N matrix input to the `B` port. The output is the solution of the equations, the M-by-N matrix *X*, and 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 the **Force input to be unit-upper triangular** check box is selected, the block replaces the elements on the diagonal of U with 1's. This is useful when matrix U is the result of another operation, such as an LDL decomposition, that uses the diagonal elements to represent the D matrix.

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

Dialog Box



Block Parameters: Backward Substitution

Backward Substitution (mask)

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

Parameters

□ Force input to be unit-upper triangular

Force input to be unit-upper triangular

Replaces the elements on the diagonal of U with 1's when selected. Tunable.

Supported Data Types

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

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

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 5-6 for related information.

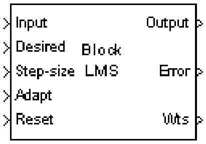
Purpose

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

Library

Filtering / Adaptive Filters

Description



Block LMS Filter

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

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

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

The weight update function for the Block LMS Filter is defined as

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

where $n = kN + i$.

The variables are as follows.

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

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

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

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

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

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

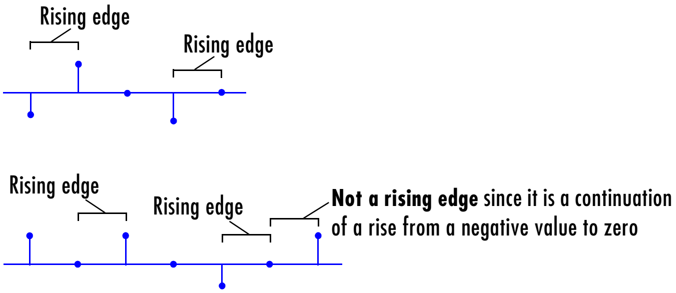
Enter the initial filter weights as a vector or a scalar in the **Initial value of filter weights** text box. If you enter a scalar, the block uses the scalar value to create a vector of filter weights. This vector has length equal to the filter length and all of its values are equal to the scalar value

If you select the **Enable/disable adaptation via input port** check box, an Adapt port appears on the block. When the input to this port is nonzero, the block continuously updates the filter weights. When the input to this port is zero, the filter weights remain at their current values.

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

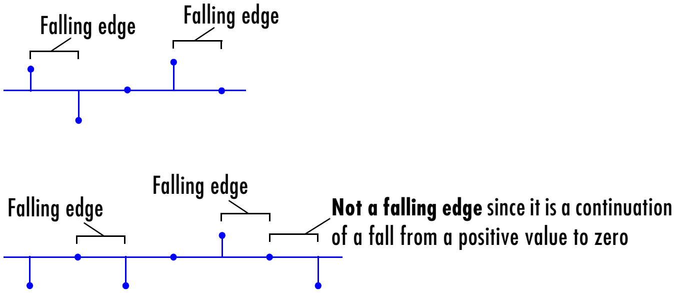
From the **Reset input** list, select `None` to disable the Reset port. To enable the Reset port, select one of the following from the **Reset input** list:

- `Rising edge` — Triggers a reset operation when the Reset input does one of the following:
  - Rises from a negative value to a positive value or zero
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure).


Rising edge
Rising edge
Rising edge
Rising edge
Not a rising edge since it is a continuation of a rise from a negative value to zero

- `Falling edge` — Triggers a reset operation when the Reset input does one of the following:
  - Falls from a positive value to a negative value or zero

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


Falling edge
Falling edge
Falling edge
Falling edge
Not a falling edge since it is a continuation
of a fall from a positive value to zero

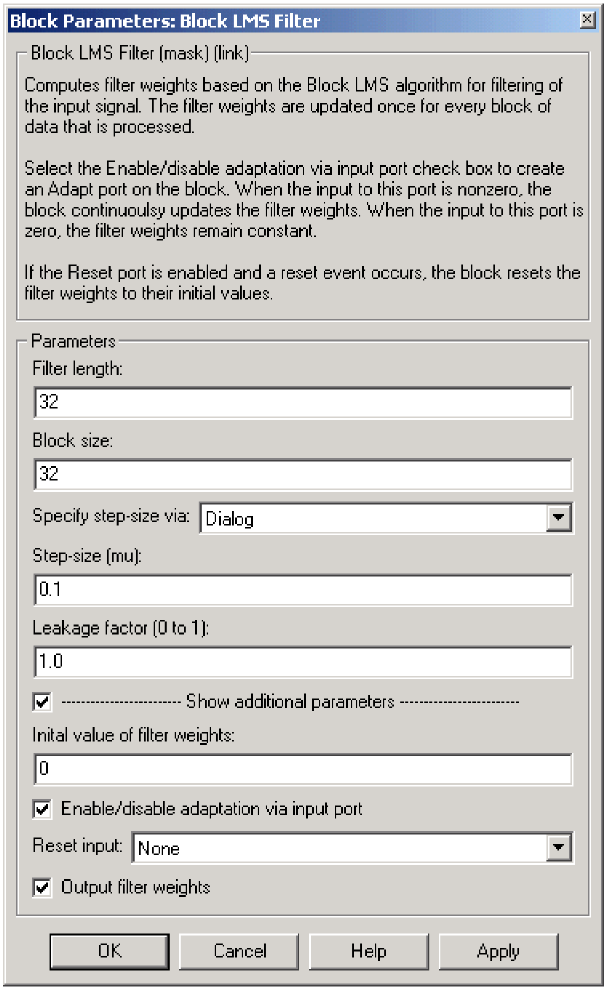
- `Either edge` — Triggers a reset operation when the Reset input is a `Rising edge` or `Falling edge` (as described above)
- `Non-zero sample` — Triggers a reset operation at each sample time that the Reset input is not zero

---

Note When running simulations in the Simulink `MultiTasking` mode, sample-based reset signals have a one-sample latency, and frame-based reset signals have one frame of latency. Thus, there is a one-sample or one-frame delay between the time the block detects a reset event, and when it applies the reset. For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" on page 2-98 and the topic called "The Simulation Parameters Dialog Box" in the Simulink documentation.

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

Dialog Box



Block LMS Filter (mask) (link)

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

Select the Enable/disable adaptation via input port check box to create an Adapt port on the block. When the input to this port is nonzero, the block continuoulsy updates the filter weights. When the input to this port is zero, the filter weights remain constant.

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

Parameters

Filter length: 32

Block size: 32

Specify step-size via: Dialog

Step-size (mu): 0.1

Leakage factor (0 to 1): 1.0

☑ ---------------------- Show additional parameters ----------------------

Inital value of filter weights: 0

☑ Enable/disable adaptation via input port

Reset input: None

☑ Output filter weights

Filter length

Enter the length of the FIR filter weights vector.

Block size

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

Specify step-size via

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

Step-size (mu)

Enter the step-size. Tunable.

Leakage factor (0 to 1)

Enter the leakage factor, $0 < 1 - \mu\alpha \leq 1$. Tunable.

Initial value of filter weights

Specify the initial values of the FIR filter weights.

Enable/disable adaptation via input port

Select this check box to enable the Adapt input port.

Reset input

Select this check box to enable the Reset input port.

Output filter weights

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

References

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

Supported Data Types

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

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

See Also

Kalman Adaptive Filter	DSP Blockset
LMS Filter	DSP Blockset
RLS Filter	DSP Blockset
Fast Block LMS Filter	DSP Blockset

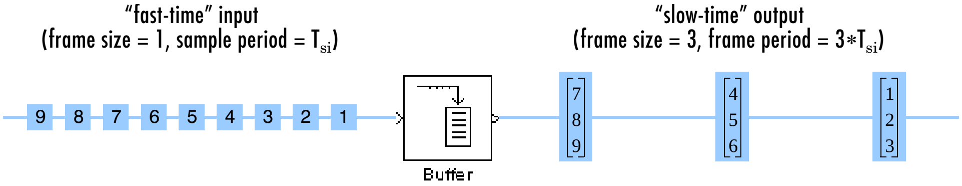
See “Adaptive Filters” on page 3-46 for related information.

Buffer

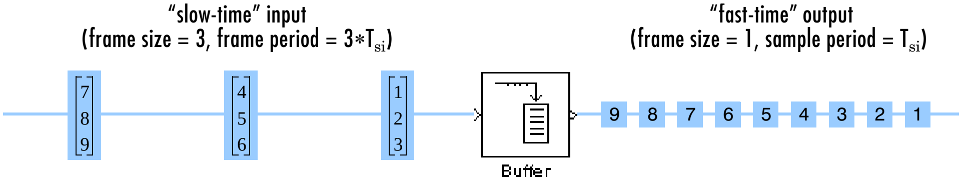
Purpose Buffer the input sequence to a smaller or larger frame size

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.


"fast-time" input
(frame size = 1, sample period = T_{si})
"slow-time" output
(frame size = 3, frame period = $3*T_{si}$)
9 8 7 6 5 4 3 2 1
Buffer
7 8 9
4 5 6
1 2 3

Buffering to a smaller frame size yields an output with a *faster* frame rate than the input, as illustrated below for scalar output.


"slow-time" input
(frame size = 3, frame period = $3*T_{si}$)
"fast-time" output
(frame size = 1, sample period = T_{si})
7 8 9
4 5 6
1 2 3
Buffer
9 8 7 6 5 4 3 2 1

The block coordinates the output *frame size* and *frame rate* of nonoverlapping buffers so that the sample period of the signal is the same at both the input and output, $T_{so} = T_{si}$.

This block supports triggered subsystems when the block's input and output rates are the same.

Sample-Based Operation

Sample-based inputs are interpreted by the Buffer block as independent channels of data. Thus, a sample-based length-N vector input is interpreted as N independent samples.

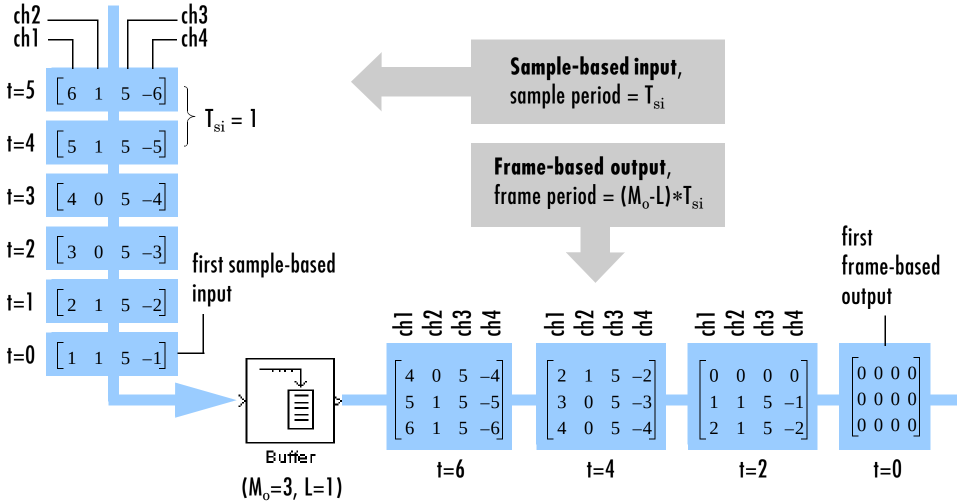
In sample-based operation, the Buffer block creates frame-based outputs from sample-based inputs. A sequence of sample-based length-N vector inputs (1-D, 2-D row, or 2-D column) is buffered into an M_o-by-N matrix, where M_o is specified by the **Output buffer size** parameter ($M_o > 1$). That is, each 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 (that is, the number of consecutive samples from the input frame to buffer into the output frame). M_o can be greater or less than the input frame size, M_i. Each of the N input channels is buffered independently.

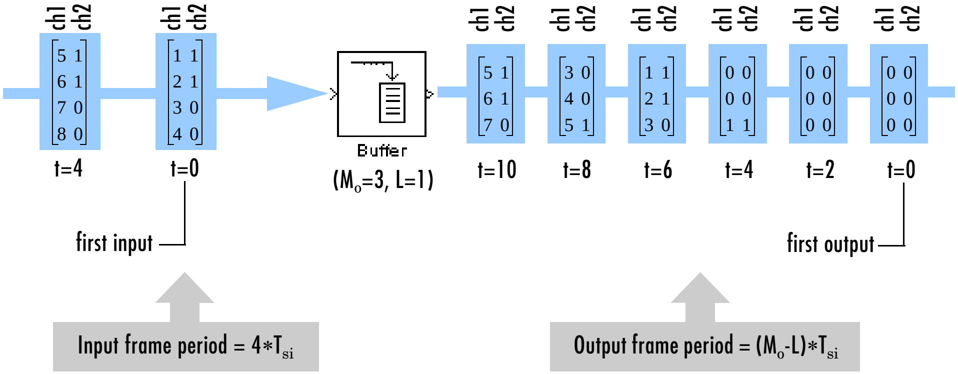
The **Buffer overlap** parameter, L, specifies the number of samples (rows) from the current output to repeat in the next output, where $L < M_o$. For $0 \leq L < M_o$, the number of *new* input samples the block acquires before propagating the buffered data to the output is the difference between the **Output buffer size** and **Buffer overlap**, M_o-L.

The input frame period is $M_i * T_{si}$, where T_{si} is the sample period. The output frame period is $(M_o\text{-}L) * T_{si}$, 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_0 output samples, or a length-M_0 vector containing the values of the first M_0 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_0-by-N matrix), **Initial conditions** can be

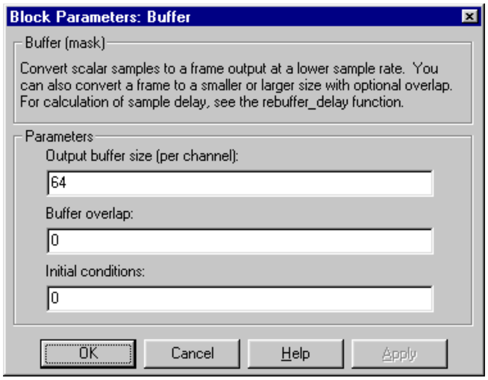
- An M_0-by-N matrix
- A length-M_0 vector to be repeated across all columns of the initial output(s)
- A scalar to be repeated across all elements of the initial output(s)

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 2-98 for more information about block rates and the Simulink tasking modes.

Dialog Box



Block Parameters: Buffer

Buffer (mask)

Convert scalar samples to a frame output at a lower sample rate. You can also convert a frame to a smaller or larger size with optional overlap. For calculation of sample delay, see the rebuffer_delay function.

Parameters

Output buffer size (per channel): 64

Buffer overlap: 0

Initial conditions: 0

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

See Also

Delay Line	DSP Blockset
Unbuffer	DSP Blockset
`rebuffer_delay`	DSP Blockset

See “Buffering Sample-Based and Frame-Based Signals” on page 2-47 for related information.

Purpose

Compute an estimate of AR model parameters using the Burg method

Library

Estimation / Parametric Estimation

Description

Burg AR
Estimator
A
G

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:

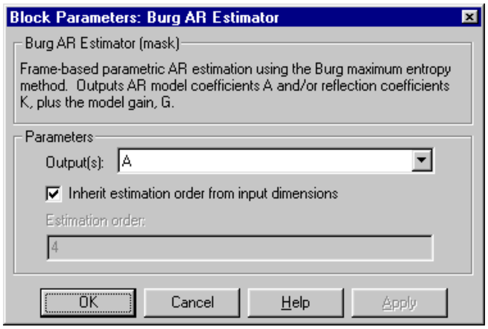
- A — The top output, A, is a column vector of length p+1 with the same frame status as the input, and contains the normalized estimate of the AR model polynomial coefficients in descending powers of z.

  ```
  [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).

Dialog Box



Block Parameters: Burg AR Estimator

Burg AR Estimator (mask)

Frame-based parametric AR estimation using the Burg maximum entropy method. Outputs AR model coefficients A and/or reflection coefficients K, plus the model gain, G.

Parameters

Output(s): A

☑ Inherit estimation order from input dimensions

Estimation order: 4

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 your data types to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-2.

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

Burg Method

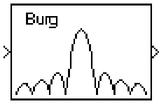
Purpose

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.

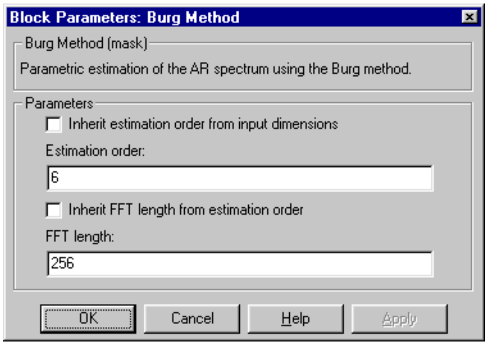
	Burg	Covariance	Modified Covariance	Yule-Walker
Characteristics	Does not apply window to data	Does not apply window to data	Does not apply window to data	Applies window to data
	Minimizes the forward and backward prediction errors in the least squares sense, with the AR coefficients constrained to satisfy 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



Block Parameters: Burg Method

Burg Method (mask)

Parametric estimation of the AR spectrum using the Burg method.

Parameters

□ Inherit estimation order from input dimensions

Estimation order: 6

□ Inherit FFT length from estimation order

FFT length: 256

Inherit estimation order from input dimensions

When selected, sets the estimation order to one less than the length of the input vector. Nontunable.

Estimation order

The order of the AR model. This parameter is enabled when **Inherit estimation order from input dimensions** is not selected. Nontunable.

Inherit FFT length from estimation order

When selected, uses the input frame size as the number of data points, N_{fft}, on which to perform the FFT. Nontunable.

FFT length

The number of data points, N_{fft}, on which to perform the FFT. 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. Nontunable.

References

Kay, S. M. *Modern Spectral Estimation: Theory and Application.* Englewood Cliffs, NJ: Prentice-Hall, 1988.

Orfanidis, J. S. *Optimum Signal Processing: An Introduction.* 2nd ed. New York, NY: Macmillan, 1985.

Supported Data Types

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

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

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 5-5 for related information.

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 when input** parameter is set to `Does not match attributes exactly`, the block generates an error only when the input possesses *none* of the attributes specified by the other parameters. Signals that possess *at least one* of the specified attributes are propagated to the output unaltered, and do not generate an error.

When the **Error when input** parameter is set to `Matches attributes exactly`, the block generates an error only when the input possesses *all* attributes specified by the other parameters. Signals that do not possess *all* of the specified attributes are propagated to the output unaltered, and do not generate an error.

Signal Attributes

The Check Signal Attributes block can test for up to five different signal attributes, as specified by the following parameters. When `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, →, and frame-based signals using a double line, ⇒.)

- **Dimensionality**

  Checks the dimension of signal for compliance (`Is...`) or noncompliance (`Is not...`) with the attributes in the subordinate **Dimension** menu, which are shown in the table below. M and N are positive integers unless otherwise indicated below.

Dimensions	Is...	Is not...
1-D	1-D vector, 1-D scalar	M-by-N matrix, 1-by-N matrix (row vector), M-by-1 matrix (column vector), 1-by-1 matrix (2-D scalar)
2-D	M-by-N matrix, 1-by-N matrix (row vector), M-by-1 matrix (column vector), 1-by-1 matrix (2-D scalar)	1-D vector, 1-D scalar
Scalar (1-D or 2-D)	1-D scalar, 1-by-1 matrix (2-D scalar)	1-D vector with length>1, M-by-N matrix with M>1 and/or N>1
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.

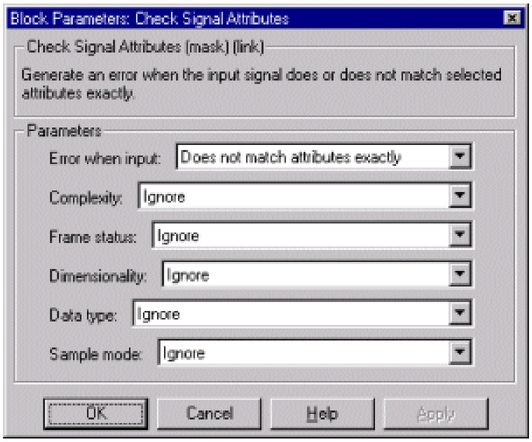
General data type	Is...	Is not...
Boolean	boolean	single, double, uint8, int8, uint16, int16, uint32, int32, fixed-point
Floating-point	single, double	boolean, uint8, int8, uint16, int16, uint32, int32, fixed-point
Fixed-point	fixed-point	boolean, uint8, int8, uint16, int16, uint32, int32, single, double
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



Block Parameters: Check Signal Attributes

Check Signal Attributes (mask) (link)

Generate an error when the input signal does or does not match selected attributes exactly.

Parameters

Error when input: Does not match attributes exactly

Complexity: Ignore

Frame status: Ignore

Dimensionality: Ignore

Data type: Ignore

Sample mode: Ignore

Error when input

Specifies whether the block generates an error when the input possesses *none* of the required attributes (`Does not match attributes exactly`), or when the input possesses *all* of the required attributes (`Matches attributes exactly`).

Complexity

The complexity for which the input should be checked, `Real` or `Complex`. If you select `Ignore` from the list, the block does not check the input's complexity.

Frame status

The frame status for which the input should be checked, `Sample-based` or `Frame-based`. If you select `Ignore` from the list, the block does not check the input's frame status.

Dimensionality

Specifies whether the input should be checked for compliance (`Is...`) or noncompliance (`Is not...`) with the attributes in the subordinate **Dimension** menu. If you select `Ignore` from the list, the block does not check the input's dimensionality.

Data type

Specifies whether the input should be checked for compliance (`Is...`) or noncompliance (`Is not...`) with the attributes in the subordinate **General data type** menu. If you select `Ignore` from the list, the block does not check the input's data type.

Sample mode

The sample mode for which the input should be checked, `Discrete` or `Continuous`. If you select `Ignore` from the list, the block does not check the input's sample 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 your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-2.

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

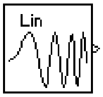
Purpose

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)
$\mathrm{f_i(t_g)}$	**Target frequency** parameter (Hz)
$\mathrm{t_g}$	**Target time** parameter (seconds)
$\mathrm{T_{sw}}$	**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.

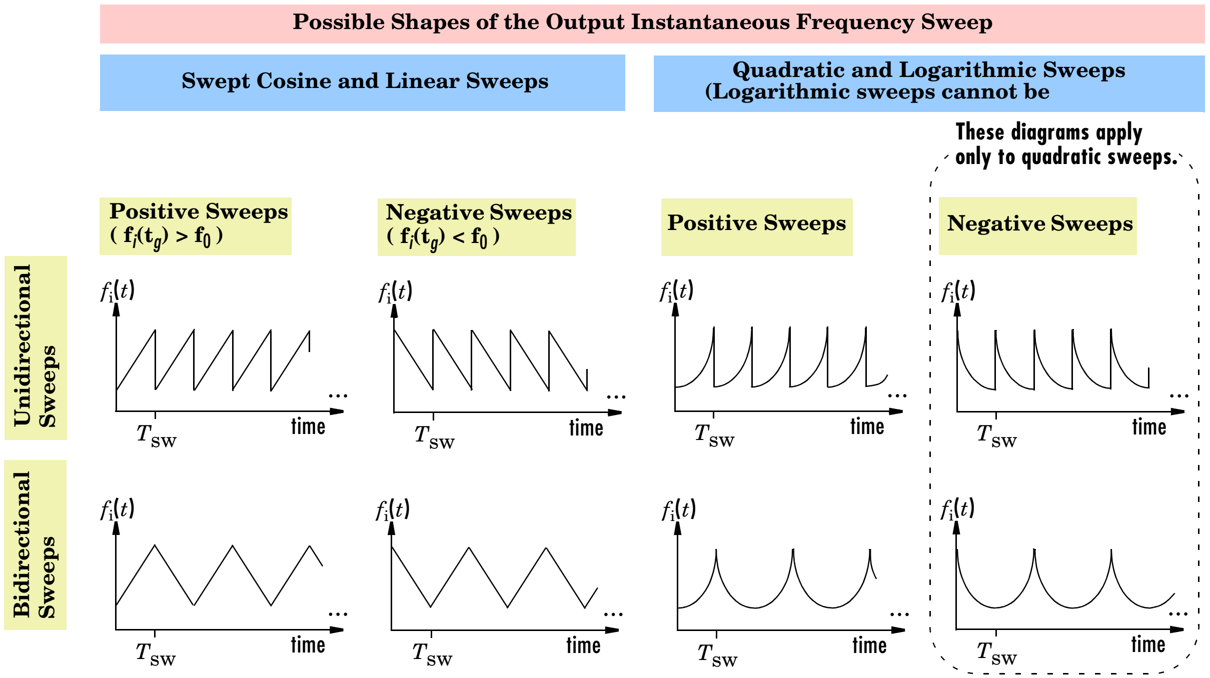
Setting of Samples Per Frame Parameter	Output Frame Status
1	Sample based
n (any integer greater than 1)	Frame based, frame size n

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" on page 7-76.

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” on page 7-77.

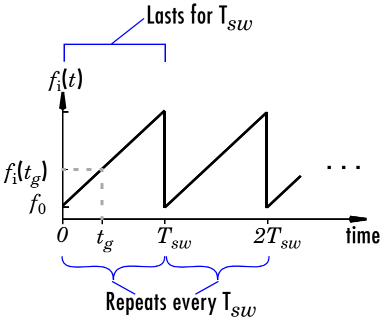
Unidirectional and Bidirectional Sweep Modes

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 7-75). The following table describes the characteristics of unidirectional and bidirectional sweeps.

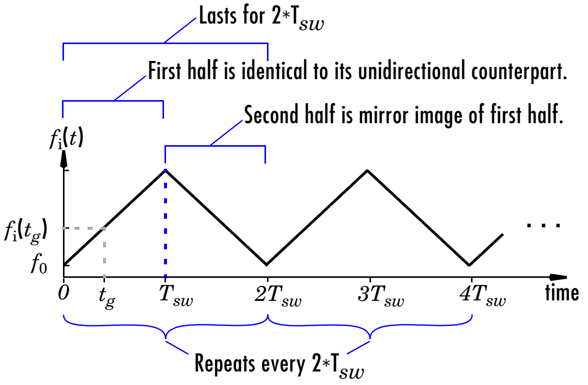
Sweep mode Parameter Settings	**Sweep Characteristics**
Unidirectional	• Lasts for one **Sweep time**, T_{sw} • Repeats once every T_{sw}
Bidirectional	• Lasts for twice the **Sweep time**, $2{*}T_{sw}$ • Repeats once every $2{*}T_{sw}$ • First half is identical to its unidirectional counterpart. • Second 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” on page 7-77.

Unidirectional Linear Sweep



Bidirectional Linear Sweep



Setting Instantaneous Frequency Sweep Values

Set the following parameters to tune the frequency values of your output frequency sweep. Note that because this is a source block, the simulation will pause while the block dialog box is open You must close the dialog box by clicking **OK** to resume the simulation.

- **Initial frequency** (Hz), f_0
- **Target frequency** (Hz), $f_i(t_g)$
- **Target time** (seconds), t_g

The following table summarizes the sweep values at specific times for all **Frequency sweep** settings. For information on the formulas used to compute sweep values at other times, see "Block Computation Methods" on page 7-78.

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" on page 7-78
- "Output Computation Method for Linear, Quadratic, and Logarithmic Frequency Sweeps" on page 7-80
- "Output Computation Method for Swept Cosine Frequency Sweep" on page 7-80

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 7-75.

The table below contains the following variables:

- $f_i(t)$ — the user-specified frequency sweep
- $f_{i(actual)}(t)$ — the actual output frequency sweep, usually equal to $f_i(t)$
- $y_{chirp}(t)$ — 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 $y_{chirp}(0) = \cos(\phi_0)$

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 $\psi(t)$
$f_i(t) = \frac{1}{2\pi} \cdot \frac{d\psi(t)}{dt}$	Phase derivative is instantaneous frequency

For instance, if you want a chirp signal with a linear instantaneous frequency sweep, you should set the **Frequency Sweep** parameter to `Linear`, and tune the linear sweep values by setting other parameters appropriately. Note that because this is a source block, the simulation will pause while the block dialog box is open. You must close the dialog box by clicking **OK** to resume the simulation. The block will output a chirp signal, the phase derivative of which is the specified linear sweep. This ensures that the instantaneous frequency of the output is the linear sweep you desired. For equations describing the linear, quadratic, and logarithmic sweeps, see "Equations for Output Computation" on page 7-78.

Output Computation Method for Swept Cosine Frequency Sweep. To generate the swept cosine chirp signal, the block sets the swept cosine chirp output as follows.

$y_{chirp}(t) = \cos(\psi(t) + \phi_0) = \cos(2\pi f_i(t)t + \phi_0)$	Swept cosine chirp output (instantaneous frequency equation, shown above, does not hold.)

Note that the instantaneous frequency equation, shown above, does not hold for the swept cosine chirp, so the user-defined frequency sweep, $f_i(t)$, is not the actual output frequency sweep, $f_{i(actual)}(t)$, 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 7-81 and "Equations for Output Computation" on page 7-78.

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 7-80. See the table called "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" on page 7-81
- "Example 2: Bidirectional Sweeps" on page 7-83
- "Example 3: When Sweep Time is Greater Than Target Time" on page 7-84

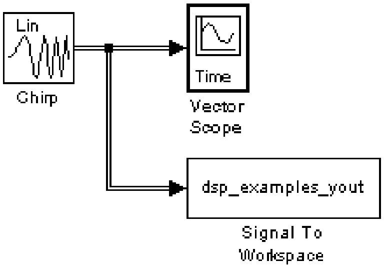
Examples 4 and 5 illustrate Chirp block settings that may produce unexpected outputs:

- "Example 4: Output Sweep with Negative Frequencies" on page 7-85
- "Example 5: Output Sweep with Frequencies Greater Than Half the Sampling Frequency" on page 7-86

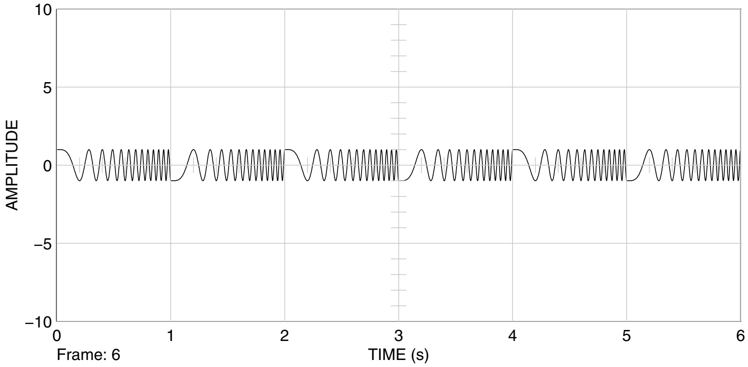
Example 1: Setting a Final Frequency Value for Unidirectional Sweeps. Often times, you may want a unidirectional sweep for which you know the initial and final frequency values. You can specify the final frequency of a unidirectional sweep by setting **Target time** equal to **Sweep time**, in which case the **Target frequency** becomes the final frequency in the sweep. The following model demonstrates this method.

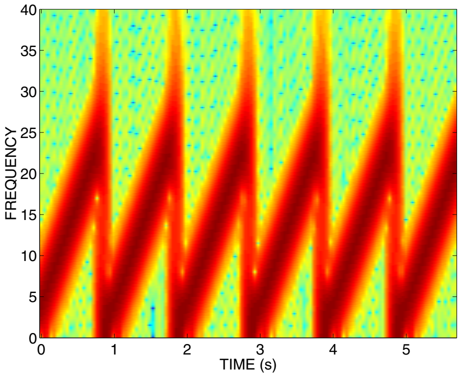
This technique may not work for swept cosine sweeps. For details, see "Cautions Regarding the Swept Cosine Sweep" on page 7-81.

Open the Example 1 model by typing `chirp_ref` at the MATLAB command line. You can also rebuild the model yourself; see the following list for model parameter settings (leave unlisted parameters in their default states).



Since **Target time** is set to equal **Sweep time** (`1` second), the **Target frequency** (`25` Hz) is the final frequency of the unidirectional sweep.





Run your model to see the time domain output, and then type the following command to view the chirp output spectrogram.

```
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

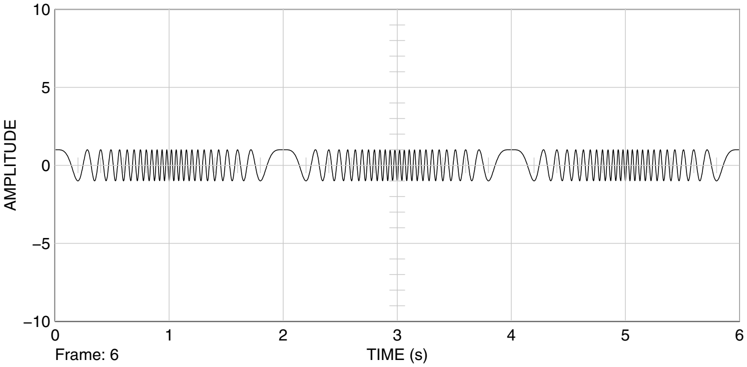
Variable name	`dsp_examples_yout`

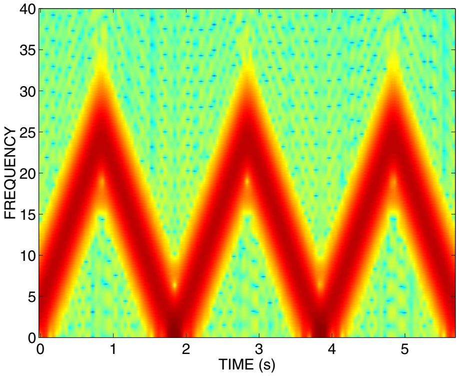
Simulation Parameters Dialog Parameters for Example 1

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.





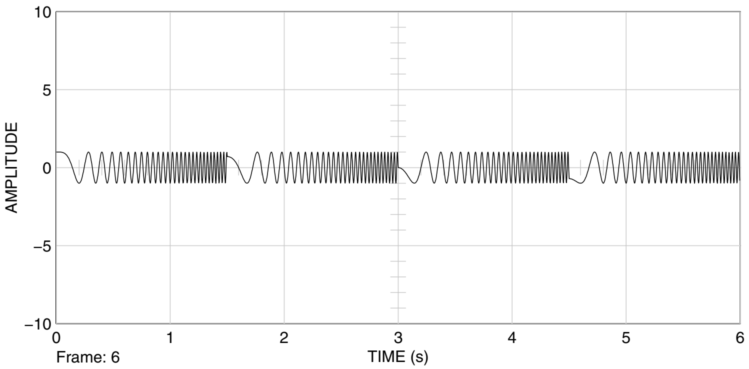
Run your model to see the time domain output, and then type the following command to view the chirp output spectrogram.

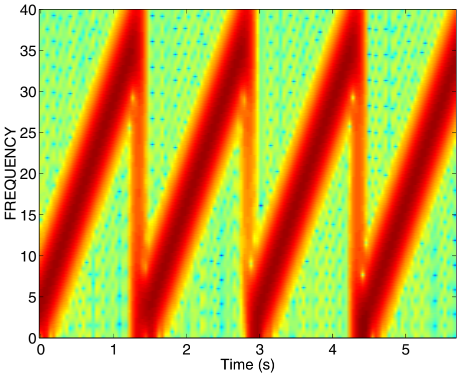
```
specgram(dsp_examples_yout,[0:.01:40],400,hamming(128),110)
```

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 Hz) at the **Target time** (1 second), but since **Sweep time** is greater than **Target time**, the sweep continues on its linear path until one **Sweep time** (1.5 seconds) is traversed.

Unexpected behavior may arise when you set **Sweep time** greater than **Target time**; see “Example 4: Output Sweep with Negative Frequencies” on page 7-85 for details.

Open the Example 3 model by clicking here in the MATLAB Help browser.





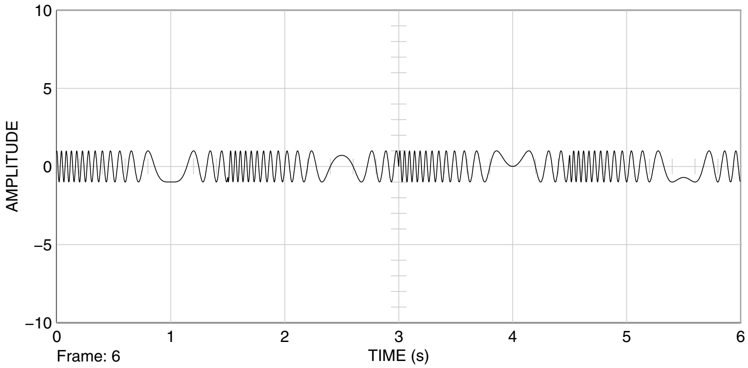
Run your model to see the time domain output, and then type the following command to view the chirp output spectrogram.

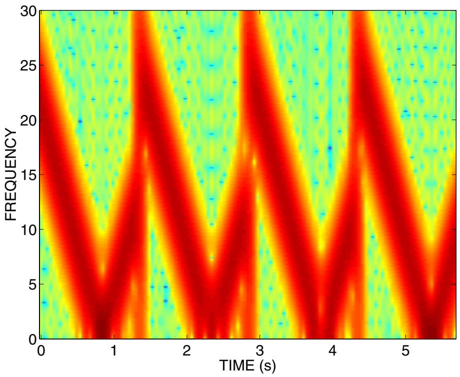
```
specgram(dsp_examples_yout,[0:.01:40],400,hamming(128),110)
```

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 Hz at one second, then continues on its negative slope, taking on negative frequency values until it traverses one **Sweep time** (1.5 seconds).

The spectrogram may reflect negative sweep frequencies along the *x*-axis so they appear to be positive. If you unexpectedly get a chirp output with a spectrogram resembling the one below, your chirp's sweep 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.



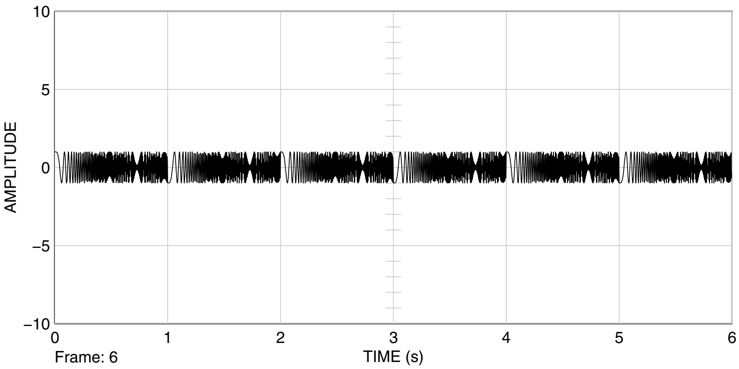


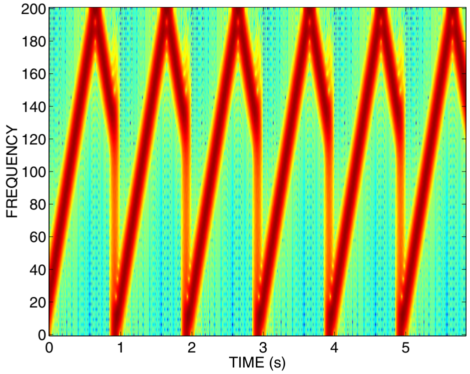
Run your model to see the time domain output, and then type the following command to view the chirp output spectrogram.

```
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 Hz), 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.

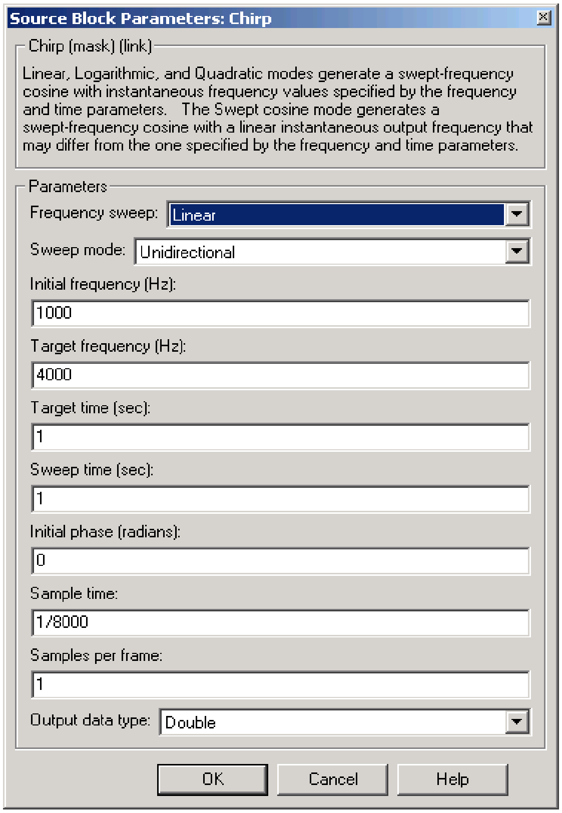




Run your model to see the time domain output, and then type the following command to view the chirp output spectrogram.

```
specgram(dsp_examples_yout,256,400,hamming(64),60)
```

Dialog Box



Source Block Parameters: Chirp

Chirp (mask) (link)

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

Parameters

Frequency sweep: Linear

Sweep mode: Unidirectional

Initial frequency (Hz): 1000

Target frequency (Hz): 4000

Target time (sec): 1

Sweep time (sec): 1

Initial phase (radians): 0

Sample time: 1/8000

Samples per frame: 1

Output data type: Double

Opening this dialog box causes a running simulation to pause. See “Changing Source Block Parameters” in the online Simulink documentation for details.

Frequency sweep

The type of output instantaneous frequency sweep, $f_i(t)$: `Linear`, `Logarithmic`, `Quadratic`, or `Swept cosine`. Tunable.

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 your data types to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-2.

See Also

Signal From Workspace	DSP Blockset
Signal Generator	Simulink
Sine Wave	DSP Blockset
`chirp`	Signal Processing Toolbox
`specgram`	Signal Processing Toolbox

Also see “Creating Signals Using Signal Generator Blocks” on page 2-35 for how to use this and other blocks to generate signals.

Cholesky Factorization

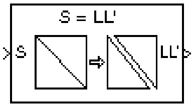
Purpose

Factor a square Hermitian positive definite matrix into triangular components.

Library

Math Functions / Matrices and Linear Algebra / Matrix Factorizations

Description

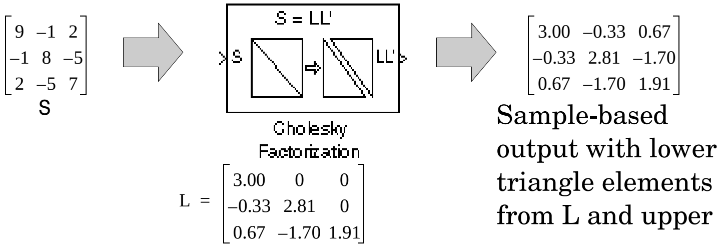


The Cholesky Factorization block uniquely factors the square Hermitian positive definite input matrix S as

$$S = LL^*$$

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:

- Hermitian — The block does *not* check whether the input is Hermitian; it uses only the diagonal and upper triangle of the input to compute the output.
- Real-valued diagonal entries — The block disregards any imaginary component of the input's diagonal entries.
- Positive definite — Set the block to notify you when the input is not positive definite as described in "Response to Nonpositive Definite Input."

Response to Nonpositive Definite Input

To generate a valid output, the block algorithm requires a positive definite input (see "Input Requirements for Valid Output" on page 7-91). Set the

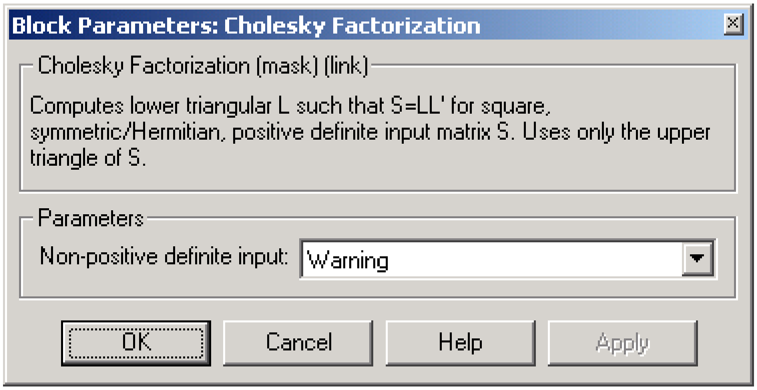
Non-positive definite input parameter to determine how the block responds to a nonpositive definite input:

- `Ignore` — Proceed with the computation and *do not* issue an alert. The output is *not* a valid factorization. A partial factorization will be present in the upper left corner of the output.
- `Warning` — Display a warning message in the MATLAB Command Window, and continue the simulation. The output is *not* a valid 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



Block Parameters: Cholesky Factorization

Cholesky Factorization (mask) (link)

Computes lower triangular L such that S=LL' for square, symmetric/Hermitian, positive definite input matrix S. Uses only the upper triangle of S.

Parameters

Non-positive definite input: Warning

Non-positive definite input

Response to nonpositive definite matrix inputs: `Ignore`, `Warning`, or `Error`. See “Response to Nonpositive Definite Input” on page 7-91. Nontunable.

References

Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

Supported Data Types

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

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

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 5-8 for related information.

Cholesky Inverse

Purpose

Compute the inverse of a Hermitian positive definite matrix using Cholesky factorization

Library

Math Functions / Matrices and Linear Algebra / Matrix Inverses

Description

Sym. Pos. Def.
Inverse
(Chol)

The Cholesky Inverse block computes the inverse of the Hermitian positive definite input matrix S by performing Cholesky factorization.

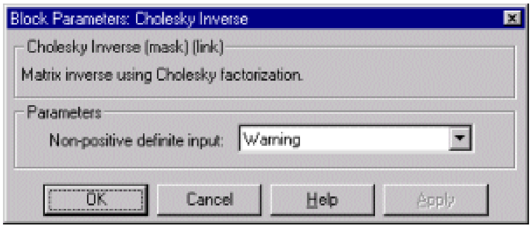
$$S^{-1} = (LL^{*})^{-1}$$

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



Block Parameters: Cholesky Inverse

Cholesky Inverse (mask) (link)

Matrix inverse using Cholesky factorization.

Parameters

Non-positive definite input: Warning

Non-positive definite input

Response to nonpositive definite matrix inputs. Nontunable.

References

Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

Supported Data Types

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

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

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 5-9 for related information.

Cholesky Solver

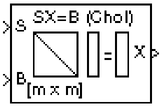
Purpose

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 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 side B is treated as an M-by-1 matrix.

Algorithm

Cholesky factorization uniquely factors the Hermitian positive definite input matrix S as

$$S = LL^*$$

where L is a lower triangular square matrix with positive diagonal elements.

The equation S*X*=B then becomes

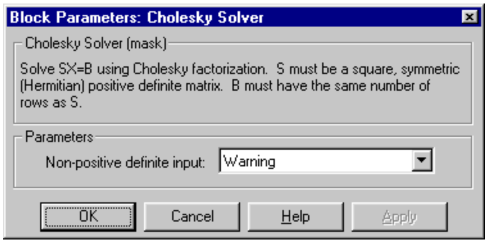
$$LL^*X = B$$

which is solved for X by making the substitution $Y = L^*X$, and solving the following two triangular systems by forward and backward substitution, respectively.

$$LY = B$$

$$L^{*}X = Y$$

Dialog Box



Block Parameters: Cholesky Solver

Cholesky Solver (mask)

Solve SX=B using Cholesky factorization. S must be a square, symmetric (Hermitian) positive definite matrix. B must have the same number of rows as S.

Parameters

Non-positive definite input: Warning

Non-positive definite input

Response to nonpositive definite matrix inputs. Nontunable.

Supported Data Types

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

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

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 5-6 for related information.

Complex Cepstrum

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 ±π radians. That is, each input channel is independently zero padded and circularly shifted to have zero phase at π radians.

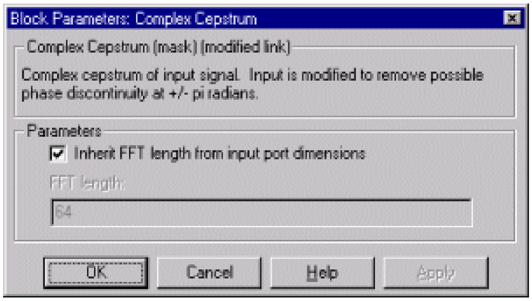
The output is a real M_o-by-N matrix, where M_o is specified by the **FFT length** parameter. Each output column contains the length-M_o complex cepstrum of the corresponding input column.

```
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 (that is, as an M-by-1 column vector), and the output is a length-M row vector. A 1-D vector input is *always* processed as a single channel, and the output is a 1-D vector.

The output is always sample based, and the output port rate is the same as the input port rate.

Dialog Box


Block Parameters: Complex Cepstrum

Complex Cepstrum (mask) (modified link)

Complex cepstrum of input signal. Input is modified to remove possible phase discontinuity at +/- pi radians.

Parameters

☑ Inherit FFT length from input port dimensions

FFT length:

64

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 your data types to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-2.

See Also

DCT	DSP Blockset
FFT	DSP Blockset
Real Cepstrum	DSP Blockset
`cceps`	Signal Processing Toolbox

Complex Exponential

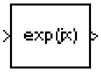
Purpose

Compute the complex exponential function

Library

Math Functions / Math Operations

Description

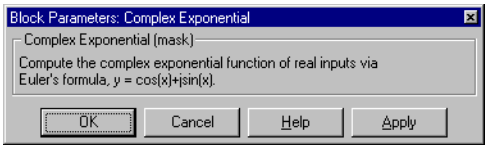


The Complex Exponential block computes the complex exponential function for each element of the real input, u.

$$y = e^{ju} = \cos u + j\sin u$$

where $j = \sqrt{-1}$. The output is complex, with the same size and frame status as the input.

Dialog Box



Complex Exponential (mask)

Compute the complex exponential function of real inputs via Euler's formula, y = cos(x)+jsin(x).

Supported Data Types

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

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

See Also

Math Function	Simulink
Sine Wave	DSP Blockset
`exp`	MATLAB

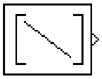
Purpose

Generate a square, diagonal matrix

Library

- DSP Sources
- Math Functions / Matrices and Linear Algebra / Matrix Operations

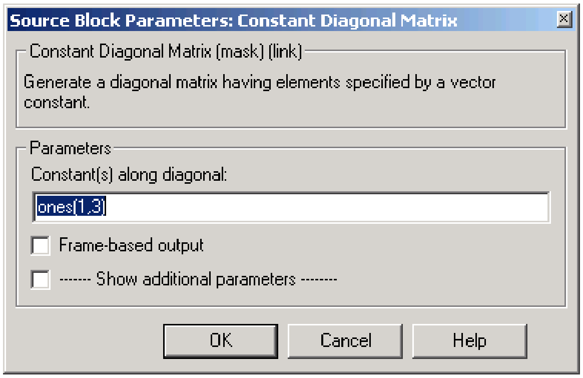
Description



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



Source Block Parameters: Constant Diagonal Matrix

Constant Diagonal Matrix (mask) (link)

Generate a diagonal matrix having elements specified by a vector constant.

Parameters

Constant(s) along diagonal:

ones(1,3)

□ Frame-based output

□ ------- Show additional parameters --------

Opening this dialog box causes a running simulation to pause. See "Changing Source Block Parameters" in the online Simulink documentation for details.

Constant(s) along diagonal

Specify the values of the elements along the diagonal. You can input a scalar or a vector. Tunable.

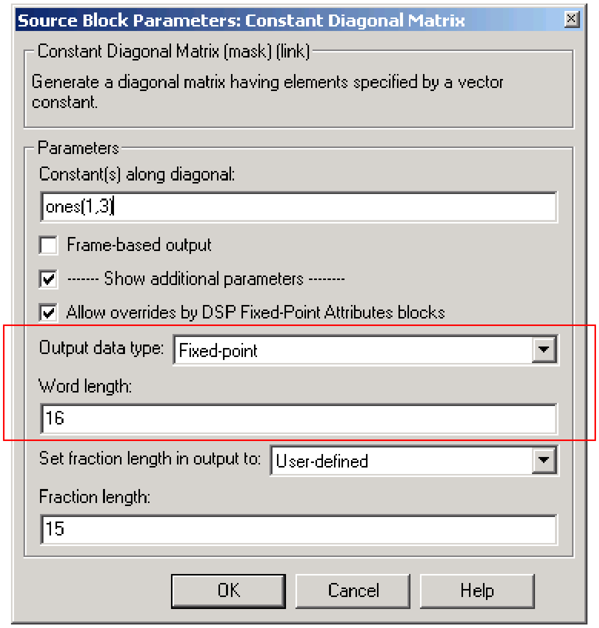
If you specify any data type information in this field, it is overridden by the value of the **Output data type** parameter.

Frame-based output

Select to cause the output of the block to be frame based. Otherwise, the output is sample based.

Show additional parameters

If selected, additional parameters specific to implementation of the block become visible as shown.



Source Block Parameters: Constant Diagonal Matrix

Constant Diagonal Matrix (mask) (link)

Generate a diagonal matrix having elements specified by a vector constant.

Parameters

Constant(s) along diagonal: ones(1,3)

□ Frame-based output

☑ ------- Show additional parameters -------

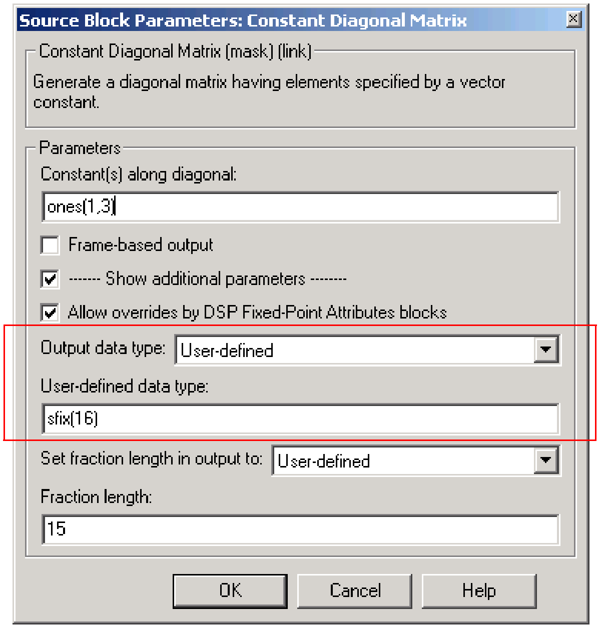
☑ Allow overrides by DSP Fixed-Point Attributes blocks

Output data type: Fixed-point

Word length: 16

Set fraction length in output to: User-defined

Fraction length: 15



Source Block Parameters: Constant Diagonal Matrix

Constant Diagonal Matrix (mask) (link)

Generate a diagonal matrix having elements specified by a vector constant.

Parameters

Constant(s) along diagonal: ones(1,3)

□ Frame-based output

☑ ------- Show additional parameters -------

☑ Allow overrides by DSP Fixed-Point Attributes blocks

Output data type: User-defined

User-defined data type: sfix(16)

Set fraction length in output to: User-defined

Fraction length: 15

Allow overrides from DSP Fixed-Point Attributes blocks

If you select this parameter, and if the **Output data type parameter** is set to `Fixed-point`, fixed-point data types for this block may be set by DSP Fixed-Point Attributes blocks in your model. If this parameter is unselected, the data types are always set by the parameters in the block mask.

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 fraction length in output to,** and **Fraction length** parameters.
- Choose `User-defined` to specify the output data type and scaling in the **User-defined data type**, **Set fraction length in output to,** and **Fraction length** parameters.
- Choose `` Inherit from `Constant(s) along diagonal' `` to set the output data type and scaling to match the values of the **Constant(s) along diagonal** parameter.
- 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 fraction length in output to

Specify the scaling of the fixed-point output by either of the following two methods:

- Choose `Best precision` to have the output scaling automatically set such that the output signal has the best possible precision.
- Choose `User-defined` to specify the output scaling in the **Fraction length** parameter.

This parameter is only visible if `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.

Fraction length

For fixed-point output data types, specify the number of fractional bits, or bits to the right of the binary point. This parameter is only visible if `Fixed-point` or `User-defined` is selected for the **Output data type** parameter, and if `User-defined` is selected for the **Set fraction length in output to** parameter.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

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

See Also

Create Diagonal Matrix	DSP Blockset
DSP Constant	DSP Blockset
Identity Matrix	DSP Blockset
`diag`	MATLAB

Also see "Creating Signals Using Constant Blocks" on page 2-32 for how to use this and other blocks to generate constant signals.

Constant Ramp

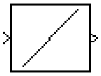
Purpose

Generate a ramp signal with length based on input dimensions

Library

Signal Operations

Description



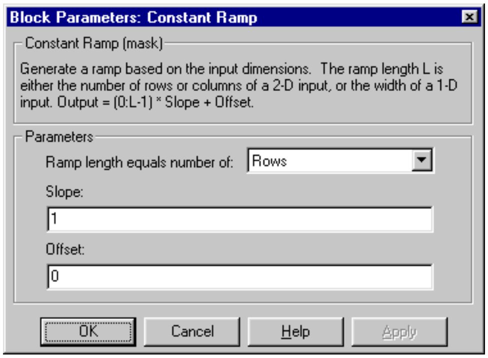
The Constant Ramp block generates the constant ramp signal

```
y = (0:L-1)*m + b
```

where `m` is the slope specified by the scalar **Slope** parameter, `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



Block Parameters: Constant Ramp

Constant Ramp (mask)

Generate a ramp based on the input dimensions. The ramp length L is either the number of rows or columns of a 2-D input, or the width of a 1-D input. Output = (0:L-1) * Slope + Offset.

Parameters

Ramp length equals number of: Rows

Slope: 1

Offset: 0

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 your data types to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-2.

See Also

Create Diagonal Matrix	DSP Blockset
Identity Matrix	DSP Blockset
DSP Constant	DSP Blockset

Also see “Creating Signals Using Constant Blocks” on page 2-32 for how to use this and other blocks to generate constant signals

Purpose

Reshape a 1-D or 2-D input to a 2-D matrix with the specified dimensions

Library

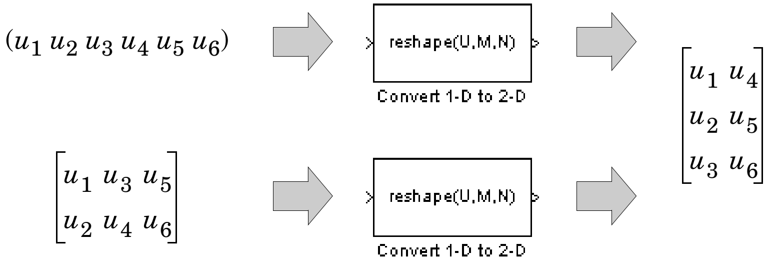
Signal Management / Signal Attributes

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
```

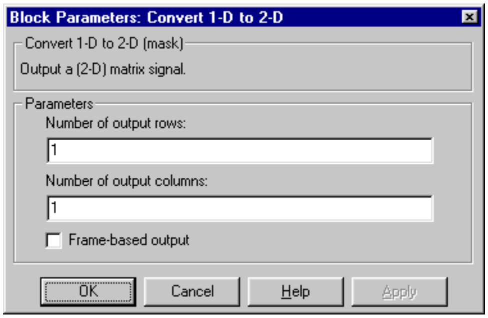
The input is reshaped *columnwise*, as shown in the two cases below. The length-6 vector and the 2-by-3 matrix are both reshaped to the same 3-by-2 output matrix.


$(u_1\ u_2\ u_3\ u_4\ u_5\ u_6)$
reshape(U,M,N)
Convert 1-D to 2-D
$\begin{bmatrix} u_1 & u_3 & u_5 \\ u_2 & u_4 & u_6 \end{bmatrix}$
reshape(U,M,N)
Convert 1-D to 2-D
$\begin{bmatrix} u_1 & u_4 \\ u_2 & u_5 \\ u_3 & u_6 \end{bmatrix}$

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



Block Parameters: Convert 1-D to 2-D

Convert 1-D to 2-D (mask)

Output a (2-D) matrix signal.

Parameters

Number of output rows:

1

Number of output columns:

1

□ Frame-based output

Number of output rows

The number of rows, M_o, in the output matrix.

Number of output columns

The number of rows, N_o, in the output matrix.

Frame-based output

Creates a frame-based output when selected.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point
- Custom data types
- Boolean
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

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

See Also

Buffer	DSP Blockset
Convert 2-D to 1-D	DSP Blockset
Frame Status Conversion	DSP Blockset
Reshape	Simulink
Submatrix	DSP Blockset

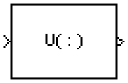
Purpose

Convert a 2-D matrix input to a 1-D vector

Library

Signal Management / Signal Attributes

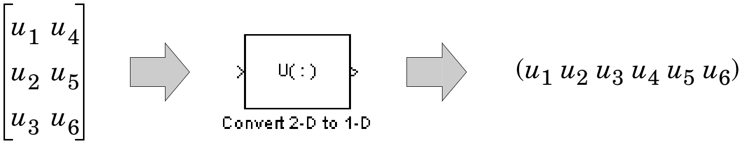
Description



The Convert 2-D to 1-D block reshapes an M-by-N matrix input to a 1-D vector with length M∗N.

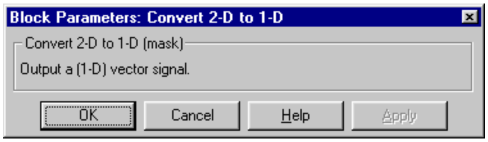
```
y = u(:)          % Equivalent MATLAB code
```

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(:)
Convert 2-D to 1-D
$(u_1\ u_2\ u_3\ u_4\ u_5\ u_6)$

The output is always sample-based.

Dialog Box



Block Parameters: Convert 2-D to 1-D

Convert 2-D to 1-D (mask)

Output a (1-D) vector signal.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point
- Custom data types
- Boolean
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

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

See Also

Buffer	DSP Blockset
Convert 1-D to 2-D	DSP Blockset
Frame Status Conversion	DSP Blockset
Reshape	Simulink
Submatrix	DSP Blockset

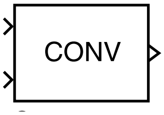
Purpose

Compute the convolution of two inputs

Library

Signal Operations

Description



The Convolution block mathematically convolves analogous columns of an M_u-by-N input matrix u and an M_v-by-N input matrix v.

The Convolution block does not accept sample-based full-dimension matrix inputs, or mixed sample-based row vector and column vector inputs. All outputs are sample based.

The Convolution block accepts both real and complex floating-point inputs. It also accepts real fixed-point inputs.

Convolving Frame-Based Inputs

Matrix inputs to the Convolution block must be frame-based. The output, y, is a frame-based (M_u+M_v-1)-by-N matrix whose jth column has elements

$$y_{i(\ ),j} = \sum_{k=1}^{max(M_u, M_v)} u_{k,j} v^{*}_{(i-k+1),j} \qquad 1 \le i \le (M_u + M_v - 1)$$

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

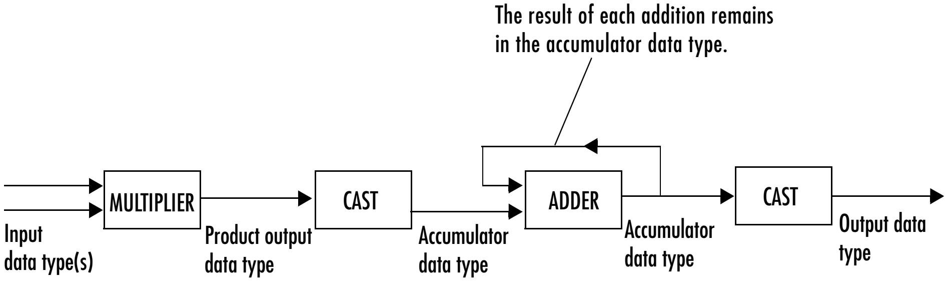
$$y_i = \sum_{k=1}^{max(M_u, M_v)} u_k v^*_{(i-k+1)} \qquad 1 \le i \le (M_u + M_v - 1)$$

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

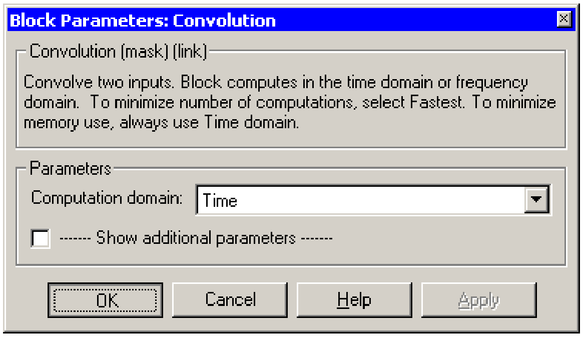
Fixed-Point Data Types

The following diagram shows the data types used within the Convolution block for fixed-point signals.



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

Dialog Box



Block Parameters: Convolution

Convolution (mask) (link)

Convolve two inputs. Block computes in the time domain or frequency domain. To minimize number of computations, select Fastest. To minimize memory use, always use Time domain.

Parameters

Computation domain: Time

□ ------ Show additional parameters ------

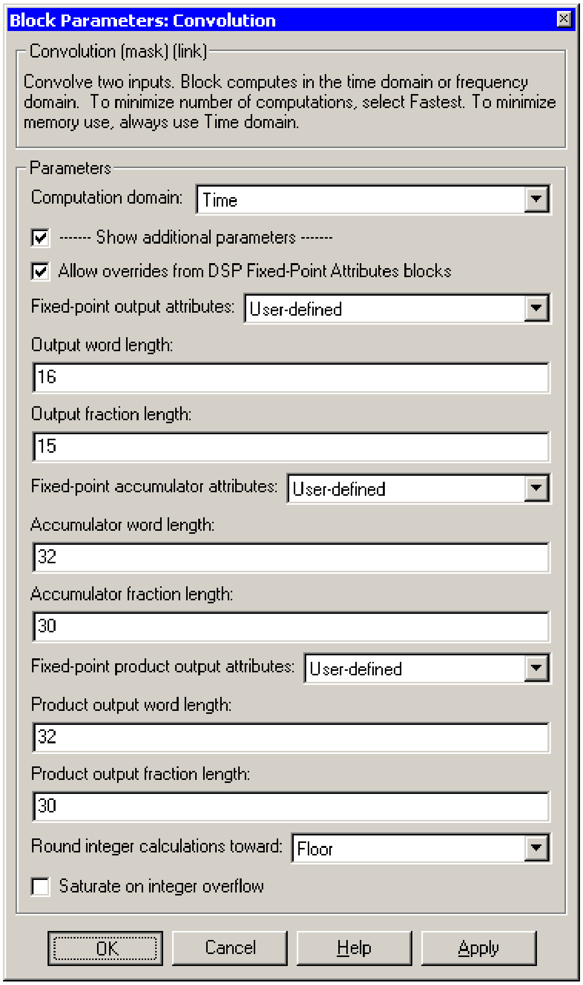
Computation domain

Set the domain in which the block computes convolutions:

- `Time` — The block computes in the time domain, which minimizes memory use
- `Frequency` — The block computes in the frequency domain, which may require fewer computations than computing in the time domain, depending on the input length
- `Fastest` — The block computes in the domain, which minimizes the number of computations

Show additional parameters

If selected, additional parameters specific to implementation of the block become visible as shown.



Block Parameters: Convolution

Convolution (mask) (link)

Convolve two inputs. Block computes in the time domain or frequency domain. To minimize number of computations, select Fastest. To minimize memory use, always use Time domain.

Parameters

Computation domain: Time

☑ ------- Show additional parameters -------

☑ Allow overrides from DSP Fixed-Point Attributes blocks

Fixed-point output attributes: User-defined

Output word length: 16

Output fraction length: 15

Fixed-point accumulator attributes: User-defined

Accumulator word length: 32

Accumulator fraction length: 30

Fixed-point product output attributes: User-defined

Product output word length: 32

Product output fraction length: 30

Round integer calculations toward: Floor

□ Saturate on integer overflow

Allow overrides from DSP Fixed-Point Attributes blocks

If you select this parameter, fixed-point data types for this block may be set by DSP Fixed-Point Attributes blocks in your model. If this parameter is unselected, the data types are always set by the parameters in the block mask.

Fixed-point output attributes

Choose how you will specify the word length and fraction length of the output of the block. If you select `Same as first input`, these characteristics will match those of the first input to the block. If you select `User-defined`, the **Output word length** and **Output fraction length** parameters become visible.

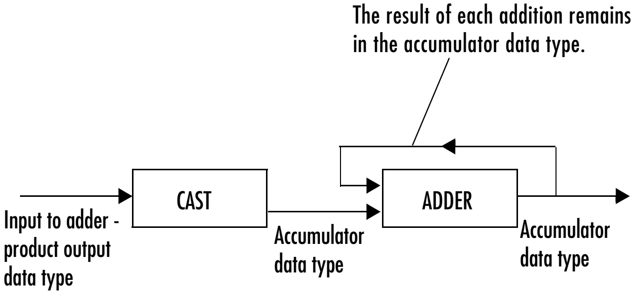
Output word length

Specify the word length, in bits, of the output. This parameter is only visible if `User-defined` is specified for the **Fixed-point output attributes** parameter.

Output fraction length

Specify the fraction length, in bits, of the output. This parameter is only visible if `User-defined` is specified for the **Fixed-point output attributes** parameter.

Fixed-point accumulator attributes


The result of each addition remains in the accumulator data type.
CAST
ADDER
Input to adder - product output data type
Accumulator data type
Accumulator data type

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

If you select `Same as first input`, the accumulator word and fraction lengths are the same as those of the first input to the block. If you select `Same as output`, they are the same as those of the output of the block. If you select `User-defined`, the **Accumulator word length** and **Accumulator fraction length** parameters become visible.

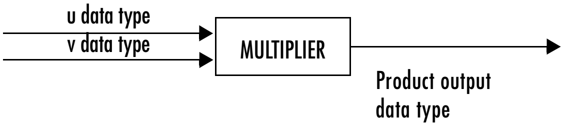
Accumulator word length

Specify the word length, in bits, of the accumulator. This parameter is only visible when `User-defined` is specified for the **Fixed-point accumulator attributes** parameter.

Accumulator fraction length

Specify the fraction length, in bits, of the accumulator. This parameter is only visible when `User-defined` is specified for the **Fixed-point accumulator attributes** parameter.

Fixed-point product output attributes



As depicted above, the output of the multiplier is placed into the product output data type and scaling. Use this parameter to specify how you would like to designate this product output word and fraction lengths.

If you select `Same as first input`, `Same as output`, or `Same as accumulator`, the product output word and fraction lengths are the same as those of the first input, output, or accumulator of the block, respectively. If you select `User-defined`, the **Product output word length** and **Product output fraction length** parameters become visible.

Product output word length

Specify the word length, in bits, of the product output. This parameter is only visible when `User-defined` is specified for the **Fixed-point product output attributes** parameter.

Product output fraction length

Specify the fraction length, in bits, of the product output. This parameter is only visible when `User-defined` is specified for the **Fixed-point product output attributes** parameter.

Round integer calculations toward

Select the rounding mode for fixed-point operations.

Saturate on integer overflow

If selected, overflows saturate.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point
- 8-, 16-, and 32-bit signed integers

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

See Also

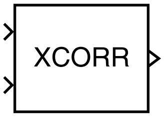
Correlation	DSP Blockset
`conv`	MATLAB

Correlation

Purpose Compute the cross-correlation of two inputs

Library Statistics

Description



The Correlation block computes the cross-correlation of analogous columns of an M_u-by-N input matrix u and an M_v-by-N input matrix v.

The frame status of both inputs to the Correlation block must be the same. The block does not accept sample-based full-dimension matrix inputs or 2-D row vector inputs. The outputs are always sample based.

The Convolution block accepts both real and complex floating-point inputs. It also accepts real fixed-point inputs.

Correlating Frame-Based Inputs

Matrix inputs to the Correlation block must be frame based. The output, y, is a frame-based (M_u+M_v-1)-by-N matrix whose jth column has elements

$$y_{(i+M_v),j} = \sum_{k=1}^{max(M_u,M_v)} u_{k,j} v^*_{(k-i),j} \qquad -M_u < i < M_v$$

where $*$ denotes the complex conjugate. Inputs u and v are zero when indexed outside of their valid ranges. When both inputs are real, the output is real; when one or both inputs are complex, the output is complex.

When one input is a column vector (single channel) and the other is a matrix (multiple channels), the single-channel input is independently cross-correlated with each channel of the multichannel input. For example, 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

The Correlation block does not support sample-based matrix inputs or 2-D row vector inputs. Therefore, all sample-based inputs are column vectors or 1-D vectors. If u and v are sample-based vectors with lengths M_u and M_v, the Correlation block performs the vector cross-correlation

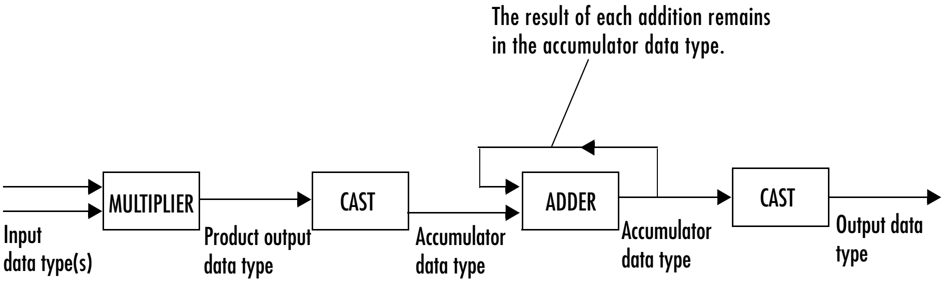
$$y_{(i+M_v)} = \sum_{k=1}^{max(M_u, M_v)} u_k v^*_{(k-i)} \qquad -M_u < i < M_v$$

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

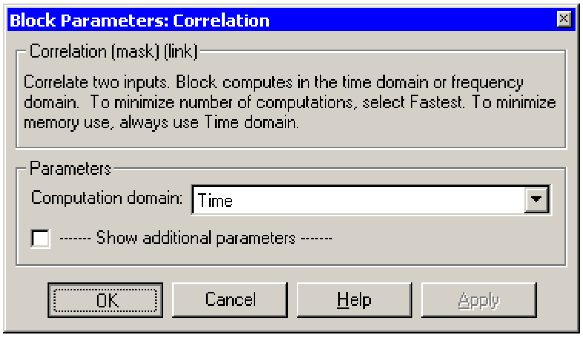
Fixed-Point Data Types

The following diagram shows the data types used within the Correlation block for fixed-point signals.



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

Dialog Box



Block Parameters: Correlation

Correlation (mask) (link)

Correlate two inputs. Block computes in the time domain or frequency domain. To minimize number of computations, select Fastest. To minimize memory use, always use Time domain.

Parameters

Computation domain: Time

☐ ------- Show additional parameters -------

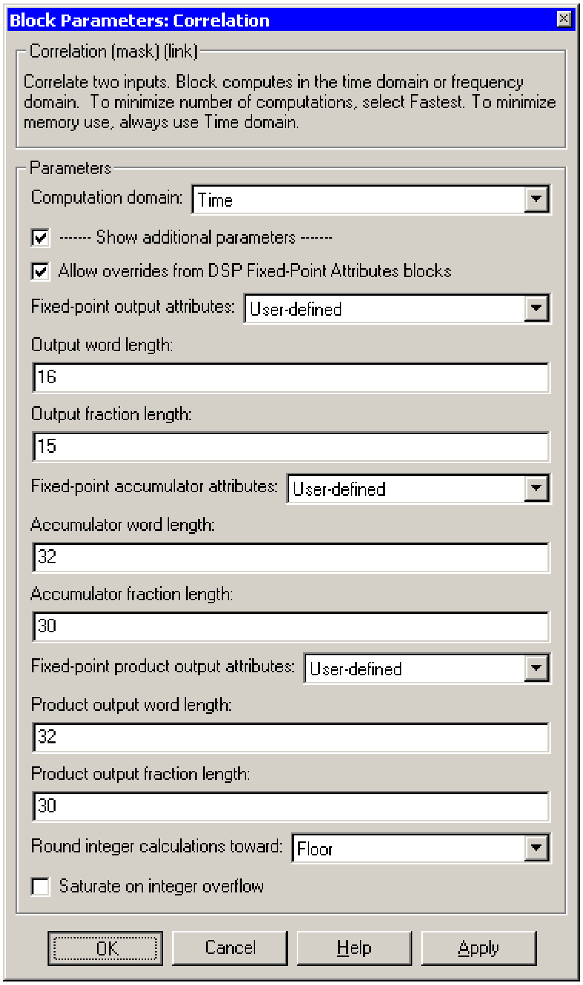
Computation domain

Set the domain in which the block computes correlations:

- `Time` — The block computes in the time domain, which minimizes memory use
- `Frequency` — The block computes in the frequency domain, which may require fewer computations than computing in the time domain, depending on the input length
- `Fastest` — The block computes in the domain, which minimizes the number of computations

Show additional parameters

If selected, additional parameters specific to implementation of the block become visible as shown.



Block Parameters: Correlation

Correlation (mask) (link)

Correlate two inputs. Block computes in the time domain or frequency domain. To minimize number of computations, select Fastest. To minimize memory use, always use Time domain.

Parameters

Computation domain: Time

☑ ------- Show additional parameters -------

☑ Allow overrides from DSP Fixed-Point Attributes blocks

Fixed-point output attributes: User-defined

Output word length: 16

Output fraction length: 15

Fixed-point accumulator attributes: User-defined

Accumulator word length: 32

Accumulator fraction length: 30

Fixed-point product output attributes: User-defined

Product output word length: 32

Product output fraction length: 30

Round integer calculations toward: Floor

□ Saturate on integer overflow

Allow overrides from DSP Fixed-Point Attributes blocks

If you select this parameter, fixed-point data types for this block may be set by DSP Fixed-Point Attributes blocks in your model. If this parameter is unselected, the data types are always set by the parameters in the block mask.

Fixed-point output attributes

Choose how you will specify the word length and fraction length of the output of the block. If you select `Same as first input`, these characteristics will match those of the first input to the block. If you select `User-defined`, the **Output word length** and **Output fraction length** parameters become visible.

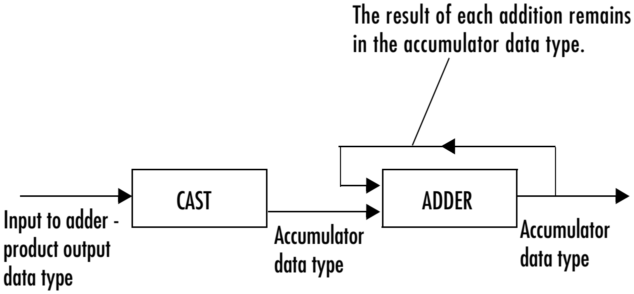
Output word length

Specify the word length, in bits, of the output. This parameter is only visible if `User-defined` is specified for the **Fixed-point output attributes** parameter.

Output fraction length

Specify the fraction length, in bits, of the output. This parameter is only visible if `User-defined` is specified for the **Fixed-point output attributes** parameter.

Fixed-point accumulator attributes


The result of each addition remains in the accumulator data type.
CAST
ADDER
Input to adder - product output data type
Accumulator data type
Accumulator data type

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

If you select `Same as first input`, the accumulator word and fraction lengths are the same as those of the input to the block. If you select `Same as output`, they are the same as those of the output of the block. If you select `User-defined`, the **Accumulator word length** and **Accumulator fraction length** parameters become visible.

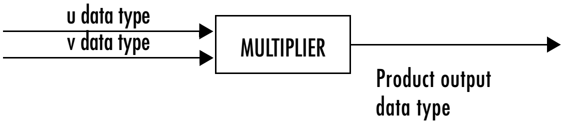
Accumulator word length

Specify the word length, in bits, of the accumulator. This parameter is only visible when `User-defined` is specified for the **Fixed-point accumulator attributes** parameter.

Accumulator fraction length

Specify the fraction length, in bits, of the accumulator. This parameter is only visible when `User-defined` is specified for the **Fixed-point accumulator attributes** parameter.

Fixed-point product output attributes



As depicted above, the output of the multiplier is placed into the product output data type and scaling. Use this parameter to specify how you would like to designate this product output word and fraction lengths.

If you select `Same as first input`, `Same as output`, or `Same as accumulator`, the product output word and fraction lengths are the same as those of the first input, output, or accumulator of the block, respectively. If you select `User-defined`, the **Product output word length** and **Product output fraction length** parameters become visible.

Product output word length

Specify the word length, in bits, of the product output. This parameter is only visible when `User-defined` is specified for the **Fixed-point product output attributes** parameter.

Product output fraction length

Specify the fraction length, in bits, of the product output. This parameter is only visible when `User-defined` is specified for the **Fixed-point product output attributes** parameter.

Round integer calculations toward

Select the rounding mode for fixed-point operations.

Saturate on integer overflow

If selected, overflows saturate.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point
- 8-, 16-, and 32-bit signed integers

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

See Also

Autocorrelation	DSP Blockset
Convolution	DSP Blockset
`xcorr`	Signal Processing Toolbox

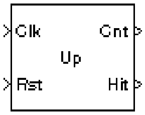
Purpose

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 (that is, 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.

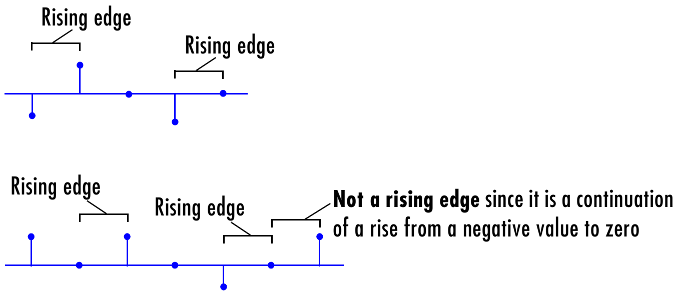
Sections of This Reference Page

- "Setting the Count Event Parameter" on page 7-125
- "Setting the Counter Size and Initial Count Parameters" on page 7-127
- "Sample-Based Operation" on page 7-127
- "Frame-Based Operation" on page 7-128
- "Free-Running Operation" on page 7-129
- "Examples" on page 7-129
- "Dialog Box" on page 7-132
- "Supported Data Types" on page 7-133
- "See Also" on page 7-134

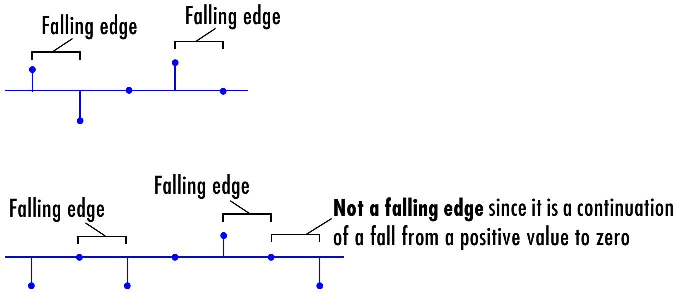
Setting the Count Event Parameter

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)


Rising edge
Rising edge
Rising edge
Rising edge
Not a rising edge since it is a continuation
of a rise from a negative value to zero

- `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)


Falling edge
Falling edge
Falling edge
Falling edge
Not a falling edge since it is a continuation
of a fall from a positive value to zero

- `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 7-129). 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 2-98.

Setting the Counter Size and Initial Count Parameters

At the start of the simulation, the block sets the counter to the value specified by the **Initial count** parameter, which can be any integer in the range defined by the **Counter size** parameter. The **Counter size** parameter allows you to choose from three standard counter ranges, or to specify an arbitrary counter limit:

- 8 bits specifies a counter with a range of 0 to 255.
- 16 bits specifies a counter with a range of 0 to 65535.
- 32 bits specifies a counter with a range of 0 to 2^{32}-1.
- User defined enables the supplementary **Maximum count** parameter, which allows you to specify an arbitrary integer as the upper count limit. The range of the counter is then 0 to the **Maximum count** value.

Sample-Based Operation

The block operates in sample-based mode when the Clk input is a sample-based scalar. Sample-based vectors and matrices are not accepted.

When the **Count direction** parameter is set to Up, a sample-based trigger event at the Clk input causes the block to increment the counter by one. The block continues incrementing the counter when triggered until the counter value reaches the upper count limit (that is 255 for an 8-bit counter). At the next Clk trigger event, the block resets the counter to 0, and resumes incrementing the counter with the subsequent Clk trigger event.

When the **Count direction** parameter is set to Down, a sample-based trigger event at the Clk input causes the block to decrement the counter by one. The block continues decrementing the counter when triggered until the counter value reaches 0. At the next Clk trigger event, the block resets the counter to the upper count limit (that is 255 for an 8-bit counter), and resumes decrementing the counter with the subsequent Clk trigger event.

Between triggering events the block holds the output at its most recent value. The block resets the counter to its initial state when the trigger event specified in the **Count event** menu is received at the optional Rst input. When trigger events are received simultaneously at the Clk and Rst ports, the block first resets the counter, and then increments or decrements appropriately. (If you do not need to reset the counter during the simulation, you can disable the Rst port by clearing the **Reset input** check box.)

The **Output** pop-up menu provides three options for the output port configuration of the block icon:

- Count configures the block icon to show a Cnt port, which produces the current value of the counter as a sample-based scalar with the same sample period as the inputs.
- Hit configures the block icon to show a Hit port. The Hit port produces zeros while the value of the counter does not equal the integer **Hit value** parameter setting. When the counter value *does* equal the **Hit value** setting, the block generates a value of 1 at the Hit port. The output is sample based with the same sample period as the inputs.
- Count and Hit configures the block icon with both ports.

Frame-Based Operation

The block operates in frame-based mode when the Clk input is a frame-based vector (that is, single channel). Multichannel frame-based inputs are not accepted.

Frame-based operation is the same as sample-based operation, except that the block increments or decrements the counter by the total number of trigger events contained in the Clk input frame. A trigger event that is split across two consecutive frames is counted in the frame that contains the conclusion of the event. When a trigger event is received at the Rst port, the block first resets the counter, and then increments or decrements the counter by the number of trigger events contained in the Clk frame.

The Cnt and Hit outputs are sample-based scalars with sample period equal to the Clk input frame period.

Free-Running Operation

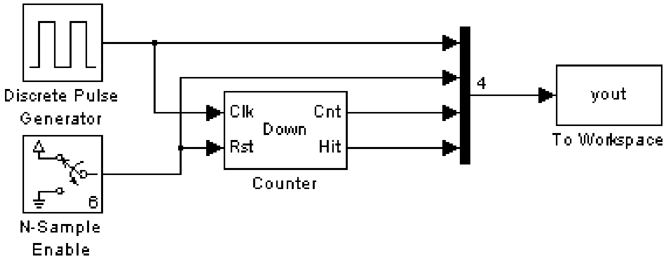
The block operates in free-running mode when `Free running` is selected from the **Count event** menu.

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 $M*T_s$.

Examples

In the model below, the `Clk` port of the Counter block is driven by the Simulink Pulse Generator block, and the `Rst` port is triggered by an N-Sample Enable block. All of the Counter block's inputs and outputs are multiplexed into a single To Workspace block using a 4-port Mux block.



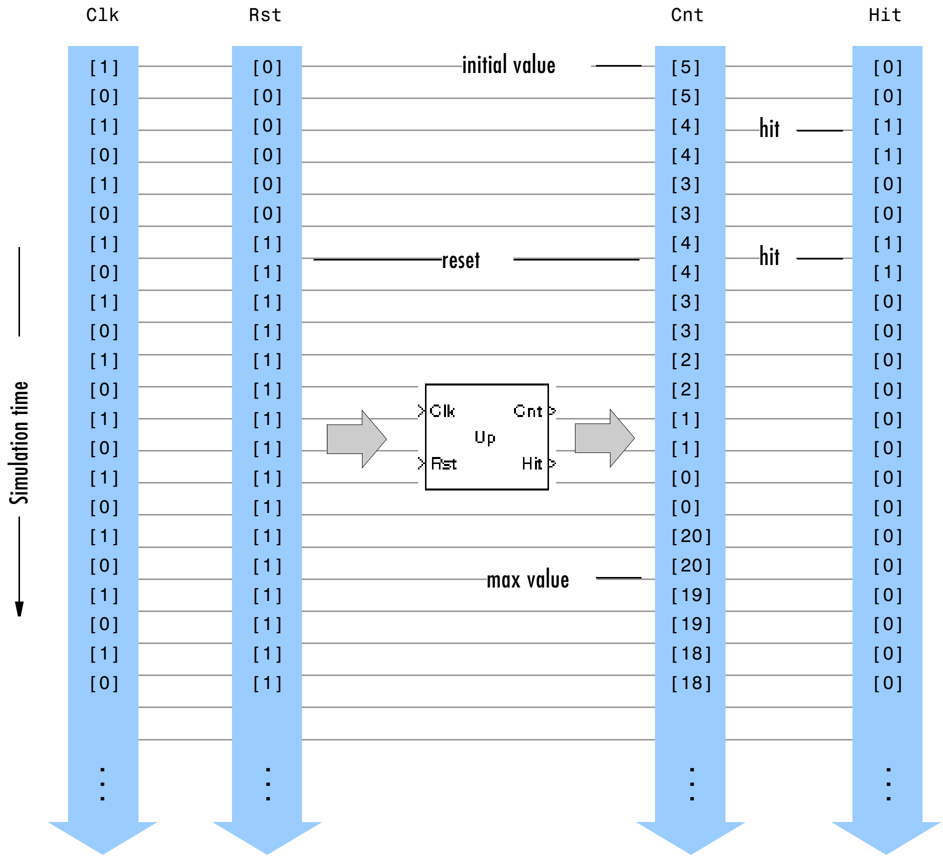
To run the model, first select **Simulation Parameters** from the **Simulation** menu, and set the **Stop time** to `30`. Then adjust the block parameters as described below. (Use the default settings for the Pulse Generator and To Workspace blocks.)

- 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** ☑
- **Count data type:** `Double`
- **Hit data type:** `Logical`

- Set the **Number of inputs** parameter of the Mux block to `4`.

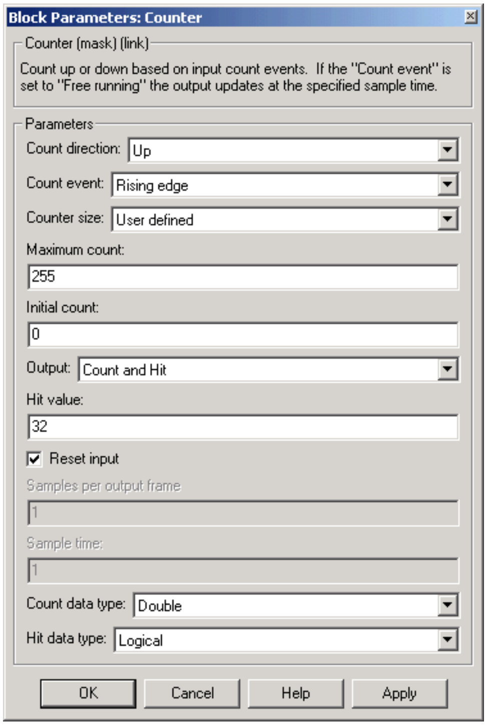
The figure below shows the first 22 samples of the model's four-column output, `yout`. The first column is the Counter block's `Clk` input, the second column is the block's `Rst` input, the third column is the block's `Cnt` output, and the fourth column is the block's `Hit` output.



Clk	Rst	Cnt	Hit
[1]	[0]	[5]	[0]
[0]	[0]	[5]	[0]
[1]	[0]	[4]	[1]
[0]	[0]	[4]	[1]
[1]	[0]	[3]	[0]
[0]	[0]	[3]	[0]
[1]	[1]	[4]	[1]
[0]	[1]	[4]	[1]
[1]	[1]	[3]	[0]
[0]	[1]	[3]	[0]
[1]	[1]	[2]	[0]
[0]	[1]	[2]	[0]
[1]	[1]	[1]	[0]
[0]	[1]	[1]	[0]
[1]	[1]	[0]	[0]
[0]	[1]	[0]	[0]
[1]	[1]	[20]	[0]
[0]	[1]	[20]	[0]
[1]	[1]	[19]	[0]
[0]	[1]	[19]	[0]
[1]	[1]	[18]	[0]
[0]	[1]	[18]	[0]
⋮	⋮	⋮	⋮

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



Block Parameters: Counter

Counter (mask) (link)

Count up or down based on input count events. If the "Count event" is set to "Free running" the output updates at the specified sample time.

Parameters

Count direction: Up

Count event: Rising edge

Counter size: User defined

Maximum count: 255

Initial count: 0

Output: Count and Hit

Hit value: 32

☑ Reset input

Samples per output frame: 1

Sample time: 1

Count data type: Double

Hit data type: Logical

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 7-125.

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 7-127.

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 the **Output** parameter is set to `Count` or `Count and Hit`.

Hit data type

The data type of the output from the `Hit` output port. For information on the `Logical` and `Boolean` options of this parameter, see "Effects of Enabling and Disabling Boolean Support" on page A-7. This parameter is available when the **Output** parameter is set to `Hit` or `Count and Hit`.

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-7. To learn how to disable Boolean output support, see “Steps to Disabling Boolean Support” on page A-8.

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

See Also

Edge Detector	DSP Blockset
N-Sample Enable	DSP Blockset
N-Sample Switch	DSP Blockset

Purpose

Compute an estimate of AR model parameters using the covariance method

Library

Estimation / Parametric Estimation

Description

Cov AR
Estimator
A
G

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.

$$H(z) = \frac{G}{A(z)} = \frac{G}{1 + a(2)z^{-1} + \ldots + a(p+1)z^{-p}}$$

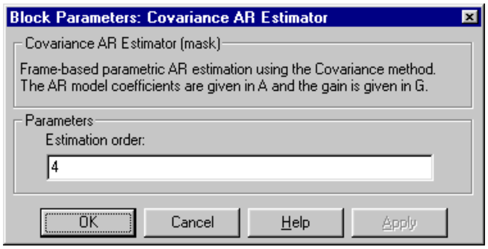
The order, p, of the all-pole model is specified by the **Estimation order** parameter. To guarantee a nonsingular output, you must set the value of p to be less than the input length. Otherwise, the output may be singular.

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



Block Parameters: Covariance AR Estimator

Covariance AR Estimator (mask)

Frame-based parametric AR estimation using the Covariance method.
The AR model coefficients are given in A and the gain is given in G.

Parameters

Estimation order:

4

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 your data types to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-2.

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

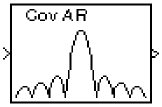
Purpose

Compute a parametric spectral estimate using the covariance method

Library

Estimation / Power Spectrum Estimation

Description



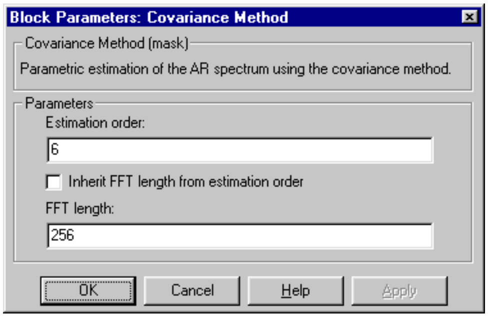
The Covariance Method block estimates the power spectral density (PSD) of the input using the covariance method. This method fits an autoregressive (AR) model to the signal by minimizing the forward prediction error in the least squares sense. The order of the all-pole model is the value specified by 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



Block Parameters: Covariance Method

Covariance Method (mask)

Parametric estimation of the AR spectrum using the covariance method.

Parameters

Estimation order:

6

☐ Inherit FFT length from estimation order

FFT length:

256

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 your data types to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-2.

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 5-5 for related information.

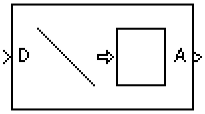
Purpose

Create a square diagonal matrix from diagonal elements

Library

Math Functions / Matrices and Linear Algebra / Matrix Operations

Description

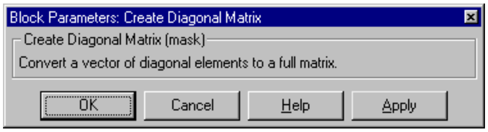


The Create Diagonal Matrix block populates the diagonal of the M-by-M matrix output with the elements contained in the length-M vector input, `D`. The elements off the diagonal are zero.

```
A = diag(D)        Equivalent MATLAB code
```

The output is always sample based.

Dialog Box



Block Parameters: Create Diagonal Matrix

Create Diagonal Matrix (mask)

Convert a vector of diagonal elements to a full 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 your data types to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-2.

See Also

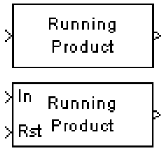
Constant Diagonal Matrix	DSP Blockset
Extract Diagonal	DSP Blockset
`diag`	MATLAB

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” on page 7-140
- “Multiplying Along Channels” on page 7-141
- “Resetting the Cumulative Product Along Channels” on page 7-143
- “Multiplying Along Columns” on page 7-144
- “Multiplying Along Rows” on page 7-145
- “Dialog Box” on page 7-146
- “Supported Data Types” on page 7-147
- “See Also” on page 7-147

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" on page 7-141
- "Multiplying Along Channels of Sample-Based Inputs" on page 7-142
- "Resetting the Cumulative Product Along Channels" on page 7-143

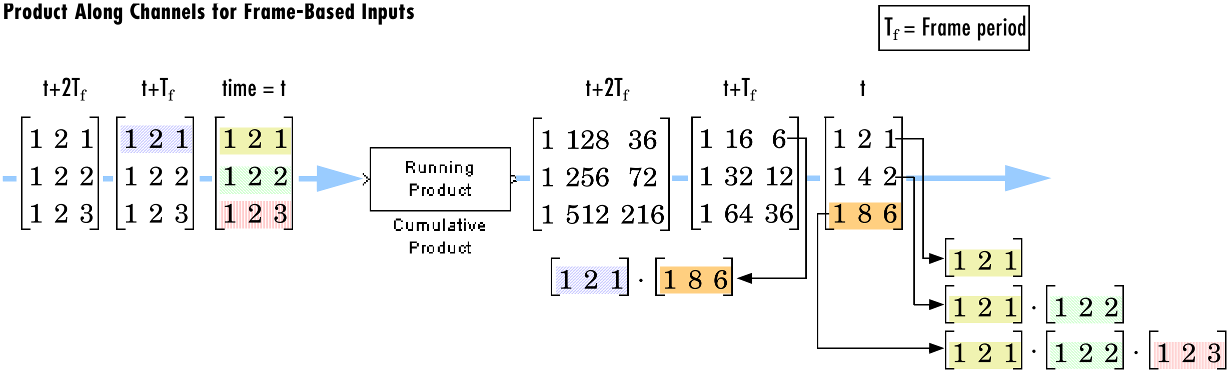
Multiplying Along Channels of Frame-Based Inputs. For frame-based inputs, the block treats each input column as an independent channel. As the following figure and equation illustrate, the output has the following characteristics:

- 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


T_f = Frame period
$t+2T_f$
$t+T_f$
time = t
$\begin{bmatrix}1&2&1\\1&2&2\\1&2&3\end{bmatrix}$ $\begin{bmatrix}1&2&1\\1&2&2\\1&2&3\end{bmatrix}$ $\begin{bmatrix}1&2&1\\1&2&2\\1&2&3\end{bmatrix}$
Running Product
Cumulative Product
$t+2T_f$
$t+T_f$
t
$\begin{bmatrix}1&128&36\\1&256&72\\1&512&216\end{bmatrix}$ $\begin{bmatrix}1&16&6\\1&32&12\\1&64&36\end{bmatrix}$ $\begin{bmatrix}1&2&1\\1&4&2\\1&8&6\end{bmatrix}$
$[1\ 2\ 1] \cdot [1\ 8\ 6]$
$[1\ 2\ 1]$
$[1\ 2\ 1] \cdot [1\ 2\ 2]$
$[1\ 2\ 1] \cdot [1\ 2\ 2] \cdot [1\ 2\ 3]$

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:

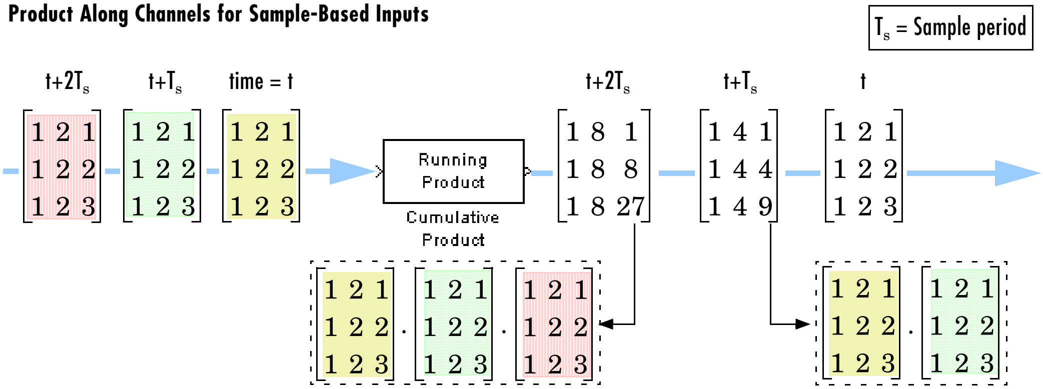
- The first output is the same as the first input.
- Each subsequent output is the element-wise product of the current input (time t) and the previous output (time $t - T_s$, where T_s is the sample period).
- The output has the same size, dimension, frame status, data type, and complexity as the input.

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

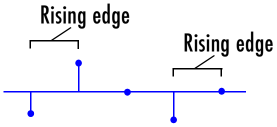


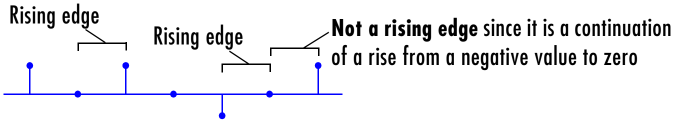
Resetting the Cumulative Product Along Channels. When you set the **Multiply input along** parameter to `Channels (running product)`, you can set the block to reset the running product whenever it detects a reset event at the optional `Rst` port. The rate of the reset signal must be a positive integer 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.

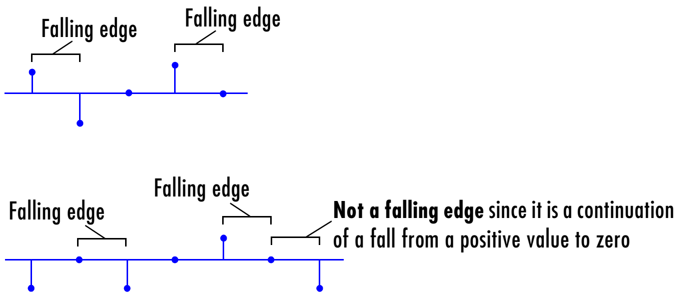
The **Reset port** parameter specifies the reset event, which can be one of the following:

- `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)




Rising edge
Rising edge
Not a rising edge since it is a continuation of a rise from a negative value to zero

- `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)


Falling edge
Falling edge
Falling edge
Falling edge
Not a falling edge since it is a continuation of a fall from a positive value to zero

- `Either edge` — Triggers a reset operation when the `Rst` input is a `Rising edge` or `Falling edge` (as described above).
- `Non-zero sample` — Triggers a reset operation at each sample time that the `Rst` input is not zero.

Note When running simulations in the Simulink `MultiTasking` mode, sample-based reset signals have a one-sample latency, and frame-based reset signals have one frame of latency. Thus, there is a one-sample or one-frame delay between the time the block detects a reset event, and when it applies the reset. For more information on latency and the Simulink tasking modes, see “Excess Algorithmic Delay (Tasking Latency)” on page 2-98.

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
```

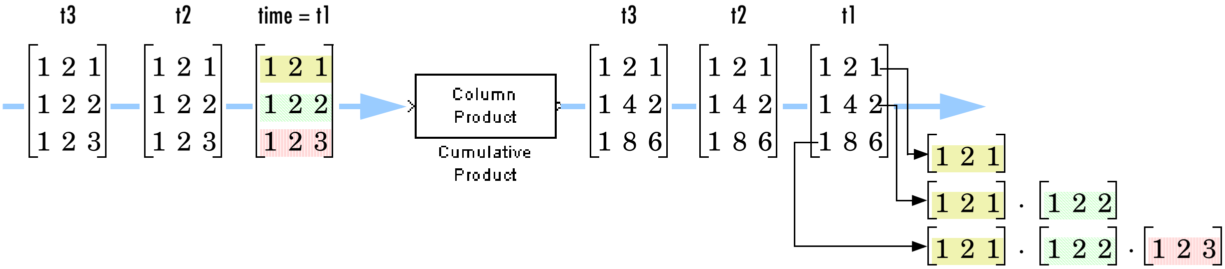
The output has the same size, dimension, frame status, data type, and complexity as the input. The *m*th output row is the element-wise product of the first *m* input rows.

Given an M-by-N input, *u*, the output, *y*, is an M-by-N matrix whose *j*th column has elements

$$y_{i,j} = \prod_{k=1}^{i} u_{k,j} \qquad 1 \le i \le M$$

The block treats length-M 1-D vector inputs as M-by-1 column vectors when multiplying along columns.

Product Along Columns



Multiplying Along Rows

When the **Multiply input along** parameter is set to `Rows`, the block computes the cumulative product of the row elements, where the current cumulative product is independent of the cumulative products of previous inputs.

```
y = cumprod(u,2)       % Equivalent MATLAB code
```

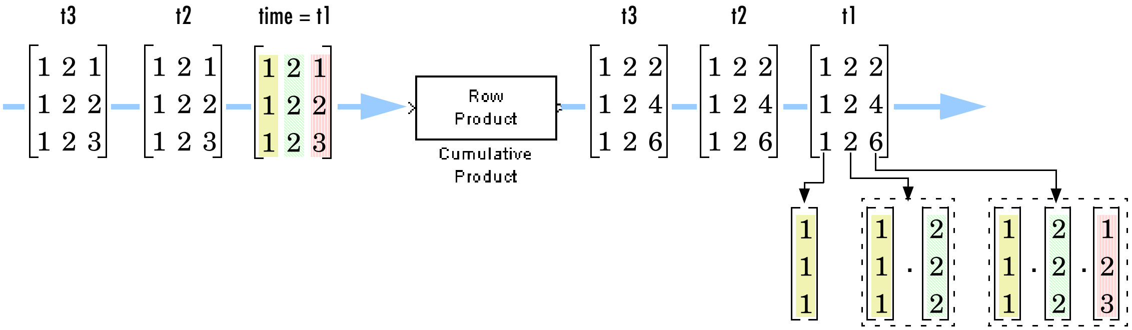
The output has the same size, dimension, frame status, and data type as the input. The *n*th output column is the element-wise product of the first *n* input columns.

Given an M-by-N input, *u*, the output, *y*, is an M-by-N matrix whose *i*th row has elements

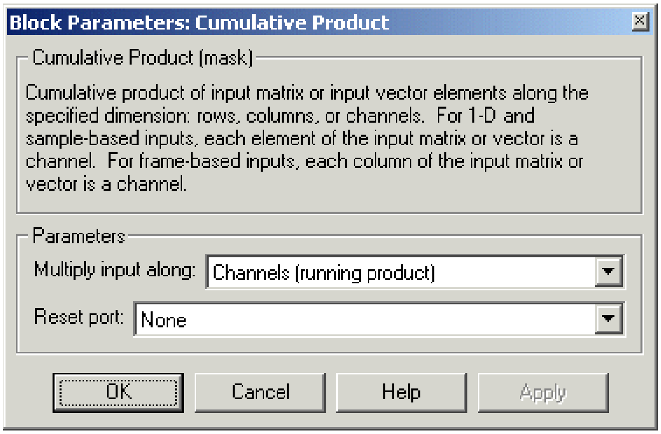
$$y_{i,j} = \prod_{k=1}^{j} u_{i,k} \qquad 1 \le j \le N$$

The block treats length-N 1-D vector inputs as 1-by-N row vectors when multiplying along rows.

Product Along Rows



Dialog Box



Block Parameters: Cumulative Product

Cumulative Product (mask)

Cumulative product of input matrix or input vector elements along the specified dimension: rows, columns, or channels. For 1-D and sample-based inputs, each element of the input matrix or vector is a channel. For frame-based inputs, each column of the input matrix or vector is a channel.

Parameters

Multiply input along: Channels (running product)

Reset port: None

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 7-141
- "Multiplying Along Columns" on page 7-144
- "Multiplying Along Rows" on page 7-145

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 7-143.

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

See Also

Cumulative Sum	DSP Blockset
Matrix Product	DSP Blockset
`cumprod`	MATLAB

Cumulative Sum

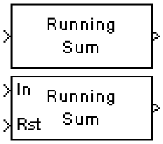
Purpose

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" on page 7-148
- "Summing Along Channels" on page 7-149
- "Resetting the Cumulative Sum Along Channels" on page 7-150
- "Summing Along Columns" on page 7-152
- "Summing Along Rows" on page 7-152
- "Dialog Box" on page 7-154
- "Supported Data Types" on page 7-155
- "See Also" on page 7-155

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 7-149
- “Summing Along Channels of Sample-Based Inputs” on page 7-150
- “Resetting the Cumulative Sum Along Channels” on page 7-150

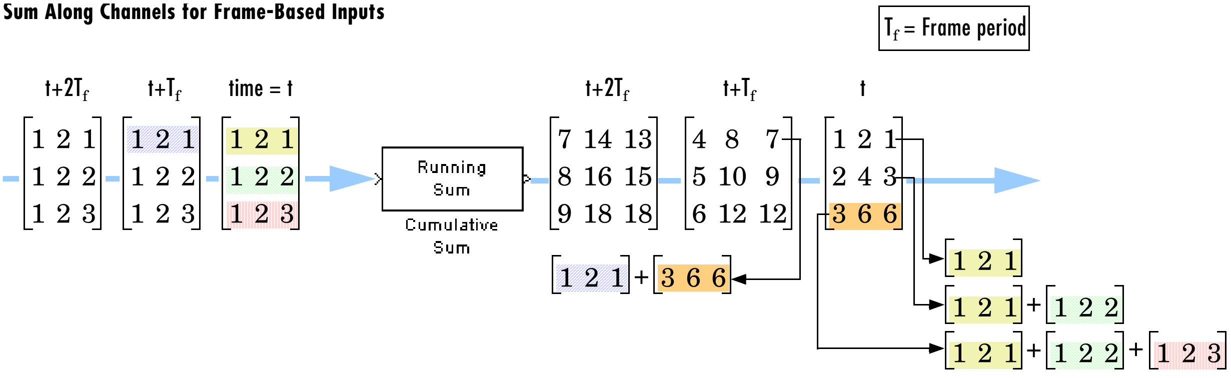
Summing Along Channels of Frame-Based Inputs. For frame-based inputs, the block treats each input column as an independent channel. As the following figure and equation illustrate, the output has the following characteristics:

- 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


T_f = Frame period
$t+2T_f$
$t+T_f$
time = t
$\begin{bmatrix}1&2&1\\1&2&2\\1&2&3\end{bmatrix}$
$\begin{bmatrix}1&2&1\\1&2&2\\1&2&3\end{bmatrix}$
$\begin{bmatrix}1&2&1\\1&2&2\\1&2&3\end{bmatrix}$
Running Sum
Cumulative Sum
$t+2T_f$
$t+T_f$
t
$\begin{bmatrix}7&14&13\\8&16&15\\9&18&18\end{bmatrix}$
$\begin{bmatrix}4&8&7\\5&10&9\\6&12&12\end{bmatrix}$
$\begin{bmatrix}1&2&1\\2&4&3\\3&6&6\end{bmatrix}$
$[1\ 2\ 1] + [3\ 6\ 6]$
$[1\ 2\ 1]$
$[1\ 2\ 1] + [1\ 2\ 2]$
$[1\ 2\ 1] + [1\ 2\ 2] + [1\ 2\ 3]$

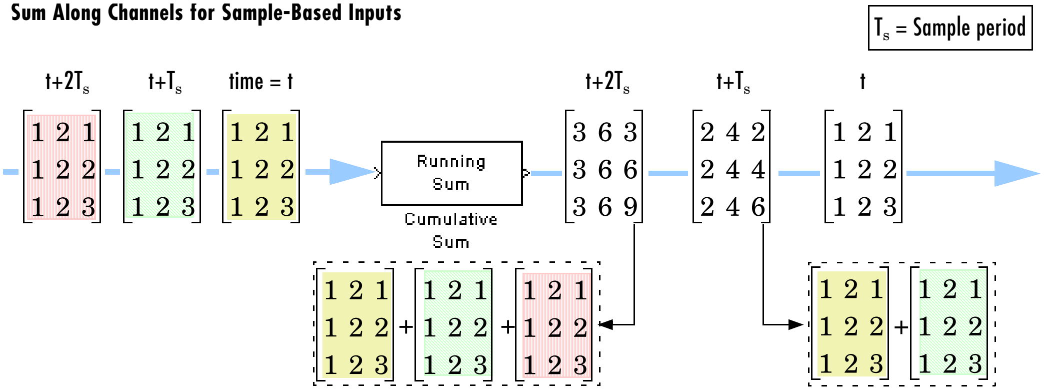
Summing Along Channels of Sample-Based Inputs. For sample-based inputs, the block treats each element of the input matrix as an independent channel. As the following figure and equation illustrate, the output has the following characteristics:

- 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

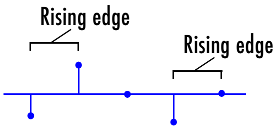


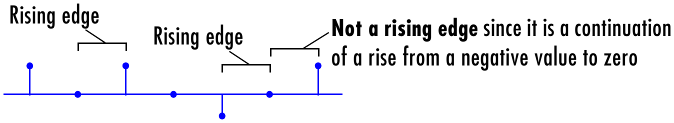
Resetting the Cumulative Sum Along Channels. When you set the **Sum input along** parameter to `Channels (running sum)`, you can set the block to reset the running sum whenever it detects a reset event at the optional `Rst` port. The rate of the reset signal must be a positive integer multiple of the 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.

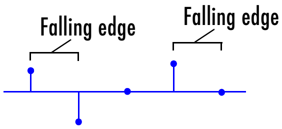
The **Reset port** parameter specifies the reset event, which can be one of the following:

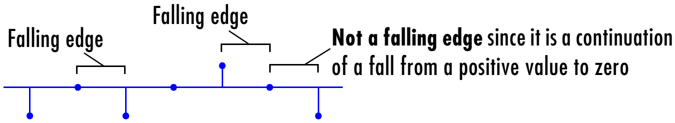
- `None` disables the `Rst` port.
- `Rising edge` — Triggers a reset operation when the `Rst` input does one of the following:
  - Rises from a negative value to a positive value or zero
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)




Rising edge
Rising edge
Not a rising edge since it is a continuation of a rise from a negative value to zero

- `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)




Falling edge
Falling edge
Not a falling edge since it is a continuation of a fall from a positive value to zero

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

- `Non-zero sample` — Triggers a reset operation at each sample time that the `Rst` input is not zero.

Note When running simulations in the Simulink `MultiTasking` mode, sample-based reset signals have a one-sample latency, and frame-based reset signals have one frame of latency. Thus, there is a one-sample or one-frame delay between the time the block detects a reset event, and when it applies the reset. For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" on page 2-98.

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 *m*th 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

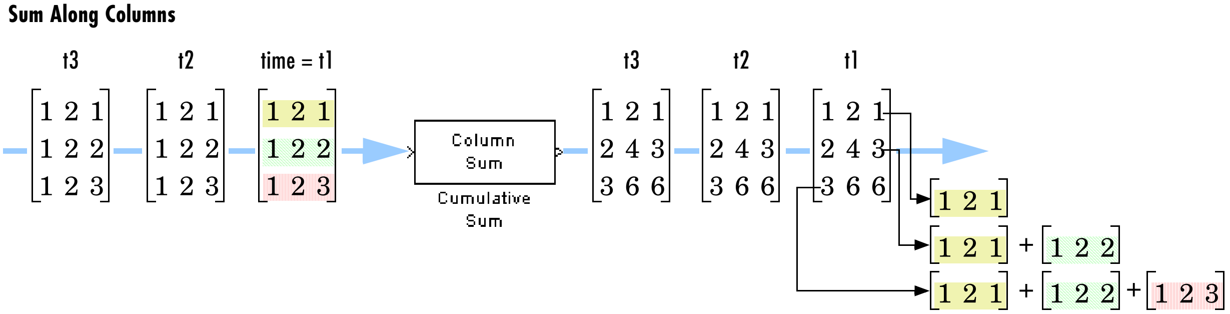
$$y_{i,j} = \sum_{k=1}^{i} u_{k,j} \qquad 1 \le i \le M$$

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
```



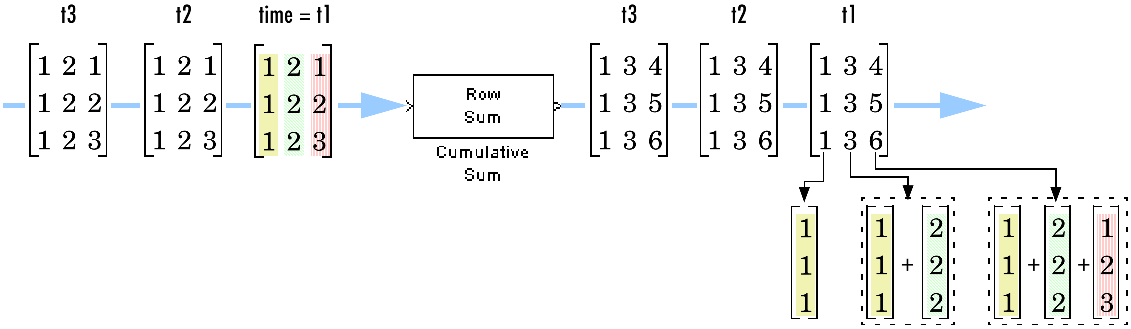
The output has the same size, dimension, frame status, and data type as the input. The *n*th output column is the sum of the first *n* input columns.

Given an M-by-N input, *u*, the output, *y*, is an M-by-N matrix whose *i*th row has elements

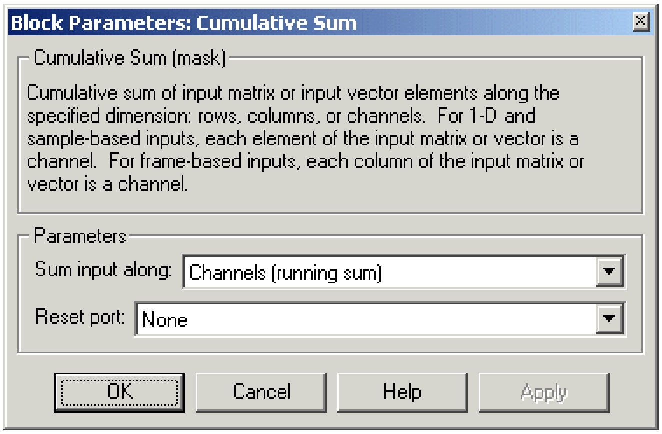
$$y_{i,j} = \sum_{k=1}^{j} u_{i,k} \qquad 1 \le j \le N$$

The block treats length-N 1-D vector inputs as 1-by-N row vectors when summing along rows.

Sum Along Rows



Dialog Box



Block Parameters: Cumulative Sum

Cumulative Sum (mask)

Cumulative sum of input matrix or input vector elements along the specified dimension: rows, columns, or channels. For 1-D and sample-based inputs, each element of the input matrix or vector is a channel. For frame-based inputs, each column of the input matrix or vector is a channel.

Parameters

Sum input along: Channels (running sum)

Reset port: None

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 7-149
- “Summing Along Columns” on page 7-152
- “Summing Along Rows” on page 7-152

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 7-150.

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

See Also

Cumulative Product	DSP Blockset
Difference	DSP Blockset
Matrix Sum	DSP Blockset
`cumsum`	MATLAB

dB Conversion

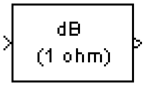
Purpose

Convert magnitude data to decibels (dB or dBm)

Library

Math Functions / Math Operations

Description



The dB Conversion block converts a linearly scaled power or amplitude input to dB or dBm. The **Input signal** parameter specifies whether the input is a power signal or a voltage signal, and the **Convert to** parameter controls the scaling of the output. When selected, the **Add eps to input to protect against "log(0) = -inf"** parameter adds a value of `eps` to all power and voltage inputs. When this option is not enabled, zero-valued inputs produce `-inf` at the output. The size and frame status of the output are the same as the input.

Power Inputs

Select `Power` as the **Input signal** parameter when the input, `u`, is a real, nonnegative, power signal (units of watts). When the **Convert to** parameter is set to `dB`, the block performs the dB conversion

```
y = 10*log10(u)            % Equivalent MATLAB code
```

When the **Convert to** parameter is set to `dBm`, the block performs the dBm conversion

```
y = 10*log10(u) + 30
```

The dBm conversion is equivalent to performing the dB operation *after* converting the input to milliwatts.

Voltage Inputs

Select `Amplitude` as the **Input signal** parameter when the input, `u`, is a real voltage signal (units of volts). The block uses the scale factor specified in ohms by the **Load resistance** parameter, `R`, to convert the voltage input to units of power (watts) before converting to dB or dBm. When the **Convert to** parameter is set to `dB`, the block performs the dB conversion

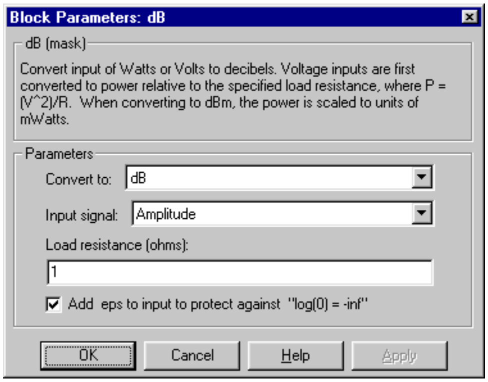
```
y = 10*log10(abs(u)^2/R)
```

When the **Convert to** parameter is set to `dBm`, the block performs the dBm conversion

```
y = 10*log10(abs(u)^2/R) + 30
```

The dBm conversion is equivalent to performing the dB operation *after* converting the `(abs(u)^2/R)` result to milliwatts.

Dialog Box



Block Parameters: dB

dB (mask)

Convert input of Watts or Volts to decibels. Voltage inputs are first converted to power relative to the specified load resistance, where P = (V^2)/R. When converting to dBm, the power is scaled to units of mWatts.

Parameters

Convert to: dB

Input signal: Amplitude

Load resistance (ohms): 1

☑ Add eps to input to protect against "log(0) = -inf"

Convert to

The logarithmic scaling to which the input is converted, `dB` or `dBm`. Tunable.

Input signal

The type of input signal, `Power` or `Amplitude`. Nontunable.

Load resistance

The scale factor used to convert voltage inputs to units of power. Tunable.

Add eps to input to protect against "log(0) = -inf"

When selected, adds `eps` to all input values (power or voltage). Tunable.

Supported Data Types

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

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

See Also

dB Gain	DSP Blockset
Math Function	Simulink
`log10`	MATLAB

dB Gain

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.

$$y_{ij} = 10u_{ij}^{(g_{ij}/k)}$$

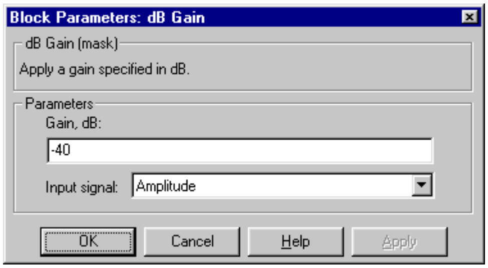
The value of k is 10 for power signals (select `Power` as the **Input signal** parameter) and 20 for voltage signals (select `Amplitude` as the **Input signal** parameter).

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



Block Parameters: dB Gain

dB Gain (mask)

Apply a gain specified in dB.

Parameters

Gain, dB: -40

Input signal: Amplitude

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 your data types to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-2.

See Also

dB Conversion	DSP Blockset
Math Function	Simulink
`log10`	MATLAB

DCT

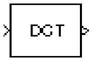
Purpose

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 *l*th column contains the length-M DCT of the corresponding input column.

$$y(k,l) = w(k)\sum_{m=1}^{M} u(m,l)\cos\frac{\pi(2m-1)(k-1)}{2M}, \qquad k = 1, \ldots, M$$

where

$$w(k) = \begin{cases} \dfrac{1}{\sqrt{M}}, & k = 1 \\ \sqrt{\dfrac{2}{M}}, & 2 \le k \le M \end{cases}$$

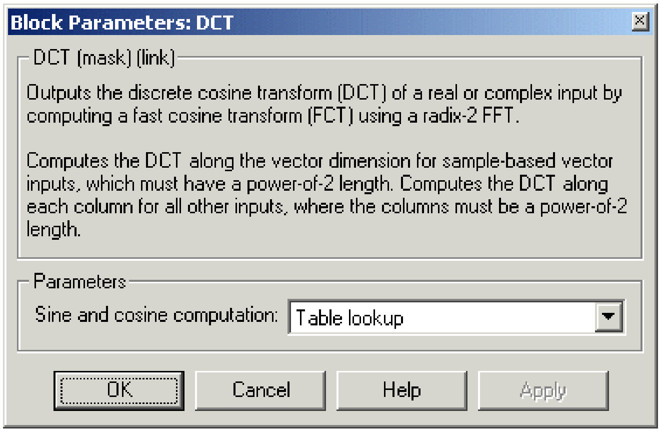
The output is always sample based, and the output port rate and data type (real/complex) are the same as those of the input port.

For convenience, length-M 1-D vector inputs and *sample-based* length-M row vector inputs are processed as single channels (that is, as M-by-1 column vectors), and the output has the same dimension as the input.

The **Sine and cosine computation** parameter determines how the block computes the necessary sine and cosine values 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



Block Parameters: DCT

DCT (mask) (link)

Outputs the discrete cosine transform (DCT) of a real or complex input by computing a fast cosine transform (FCT) using a radix-2 FFT.

Computes the DCT along the vector dimension for sample-based vector inputs, which must have a power-of-2 length. Computes the DCT along each column for all other inputs, where the columns must be a power-of-2 length.

Parameters

Sine and cosine computation: Table lookup

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 your data types to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-2.

See Also

Complex Cepstrum	DSP Blockset
FFT	DSP Blockset
IDCT	DSP Blockset
Real Cepstrum	DSP Blockset
`dct`	Signal Processing Toolbox

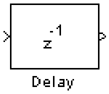
Purpose

Delay the discrete-time input by a specified number of samples or frames

Library

Signal Operations

Description



The Delay block delays a discrete-time input by the number of samples or frames specified in the **Delay units** and **Delay** parameters. The **Delay** value must be an integer value greater than or equal to zero. Also, if you enter a value of zero for the **Delay** parameter, any initial conditions you might have entered have no effect on the output.

The Delay block allows you to set the initial conditions of the signal that is being delayed. The initial conditions must be numeric. Select the **Show additional parameters** check box in order to specify the initial conditions.

This block reference contains the following topics:

- **Sample-Based Operation** — Use the Delay block with a sample-based input signal
- **Frame-Based Operation** — Use the Delay block with a frame-based input signal

Sample-Based Operation

When the input is a sample-based M-by-N matrix, where $M \geq 1$ and $N \geq 1$,the block treats each of the M∗N matrix elements as an independent channel.

If the input is a sample-based scalar, the **Delay** parameter can be a scalar integer by which to equally delay all channels. If the input is a sample-based vector, the **Delay** parameter can be a scalar integer by which to equally delay all channels, or a vector whose length is equal to the number of channels. If the input is a sample-based M-by-N matrix, where M>1 and N>1, then the **Delay** parameter can be a scalar integer by which to equally delay all channels or an M-by-N matrix of nonnegative integers that specify the number of sample intervals to delay each channel of the input.

There are four different choices for initial conditions. The initial conditions can be the same or different for each channel. They can also be the same or different along each channel. The next sections describe the behavior of the block for each of these four cases:

- "Case 1 — Use the Same Initial Conditions for Each Channel and Within a Channel" on page 7-164
- "Case 2 — Use Different Initial Conditions for Each Channel and the Same Initial Conditions Within a Channel" on page 7-165
- "Case 3 — Use the Same Initial Conditions for Each Channel and Different Initial Conditions Within a Channel" on page 7-165
- "Case 4 — Use Different Initial Conditions for Each Channel and Within a Channel" on page 7-166

Case 1 — Use the Same Initial Conditions for Each Channel and Within a Channel

Enter a scalar value for the initial conditions. This value is used as the constant initial condition value for each of the channels.

For example, suppose your input is a sample-based matrix.

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}, \begin{bmatrix} 3 & 3 \\ 3 & 3 \end{bmatrix}, \ldots$$

You want the initial conditions of your four-channel signal to be identical and zero for the first two samples:

1. For the **Delay (samples)** parameter, type 2.
2. Clear the **Specify different initial conditions for each channel** and **Specify different initial conditions within a channel** check boxes.
3. For the **Initial conditions** parameter, specify a scalar value of 0.

   The output of the delay block is

   $$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}, \begin{bmatrix} 3 & 3 \\ 3 & 3 \end{bmatrix}, \ldots$$

   Note how 0, the scalar initial condition value, is used for each channel and within the channels. It is the output at sample time zero and sample time one.

Case 2 — Use Different Initial Conditions for Each Channel and the Same Initial Conditions Within a Channel

The initial conditions can be either a matrix for matrix input or a vector for vector input. These initial condition values are used as the constant initial condition value for each of the channels.

For example, suppose your input is a sample-based matrix.

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}, \begin{bmatrix} 3 & 3 \\ 3 & 3 \end{bmatrix}, \ldots$$

You want the initial conditions of your four-channel signal to be

$$\begin{bmatrix} 7 & 9 \\ 11 & 13 \end{bmatrix}$$

for the first two samples:

1. For the **Delay (samples)** parameter, type `2`.
2. Select the **Specify different initial conditions for each channel** check box.
3. Clear the **Specify different initial conditions within a channel** check box.
4. For the **Initial conditions** parameter, type `[7 9; 11 13]`.

   The output of the delay block is

   $$\begin{bmatrix} 7 & 9 \\ 11 & 13 \end{bmatrix}, \begin{bmatrix} 7 & 9 \\ 11 & 13 \end{bmatrix}, \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}, \begin{bmatrix} 3 & 3 \\ 3 & 3 \end{bmatrix}, \ldots$$

   Note how the initial condition matrix is the output at sample time zero and sample time one. Different initial conditions are used for each channel; the same initial condition value is used within a channel.

Case 3 — Use the Same Initial Conditions for Each Channel and Different Initial Conditions Within a Channel

In this case, if the input is a sample-based vector, the **Delay** parameter can be a scalar integer by which to equally delay all channels or a vector whose length is equal to the number of channels. All the values of this vector must be equal.

Enter the initial conditions as a vector, where the vector length is equal to the delay value. These values are used as the initial condition value along each of the channels to be delayed.

For example, suppose your input is a sample-based matrix.

$$\begin{bmatrix}1 & 1\\1 & 1\end{bmatrix}, \begin{bmatrix}2 & 2\\2 & 2\end{bmatrix}, \begin{bmatrix}3 & 3\\3 & 3\end{bmatrix}, \ldots$$

You want the initial conditions of your four channel signal to be the same along each of the channels to be delayed:

1. For the **Delay (samples)** parameter, type `2`.
2. Clear the **Specify different initial conditions for each channel** check box.
3. Select the **Specify different initial conditions within a channel** check box.
4. For the **Initial conditions** parameter, type `[10 20]`.

   The output of the delay block is

   $$\begin{bmatrix}10 & 10\\10 & 10\end{bmatrix}, \begin{bmatrix}20 & 20\\20 & 20\end{bmatrix}, \begin{bmatrix}1 & 1\\1 & 1\end{bmatrix}, \begin{bmatrix}2 & 2\\2 & 2\end{bmatrix}, \begin{bmatrix}3 & 3\\3 & 3\end{bmatrix}, \ldots$$

   Note how the first element of the initial conditions vector is the output, for all channels, at sample time zero. The second element of the initial conditions vector is the output, for all channels, at sample time one. The same initial conditions are used for each channel, but different initial condition values are used with a channel.

Case 4 — Use Different Initial Conditions for Each Channel and Within a Channel

Enter a cell array for your initial condition values. Each cell of the cell array represents the delay values for one channel. The cell array must have the same size as your input signal. Or, if you have a nonmatrix input and a scalar delay value, you can enter the initial conditions as a matrix.

For example, suppose your input is a sample-based vector.

$$\begin{bmatrix}1 & 1\end{bmatrix}, \begin{bmatrix}2 & 2\end{bmatrix}, \begin{bmatrix}3 & 3\end{bmatrix}, \ldots$$

You want the initial conditions of your two channel signal to be different for each channel and along each channel:

1 For the **Delay (samples)** parameter, type `2`.
2 Select the **Specify different initial conditions for each channel** and **Specify different initial conditions within a channel** check boxes.
3 For the **Initial conditions** parameter, type `[10 20; 30 40]`

   The output of the delay block is

   $$\begin{bmatrix}10 & 20\end{bmatrix}, \begin{bmatrix}30 & 40\end{bmatrix}, \begin{bmatrix}1 & 1\end{bmatrix}, \begin{bmatrix}2 & 2\end{bmatrix}\cdots$$

   Note that the first row of the initial conditions vector is the output at sample time zero. The second row of the initial conditions vector is the output at sample time one. Different initial conditions are used for each channel and within the channels.

In addition, suppose your input is a sample-based matrix.

$$\begin{bmatrix}1 & 1\\1 & 1\end{bmatrix}, \begin{bmatrix}2 & 2\\2 & 2\end{bmatrix}, \begin{bmatrix}3 & 3\\3 & 3\end{bmatrix}, \ldots$$

You want the initial conditions of your two-channel signal to be different for each channel and along each channel.

1 For the **Delay (samples)** parameter, type `2`.
2 Select the **Specify different initial conditions for each channel** and the **Specify different initial conditions within a channel** check boxes.
3 For the **Initial conditions** parameter, type `{[11 15] [12 16]; [13 17] [14 18]}`. Note that the dimensions of the cell array match the dimensions of the input. Also, each element of the cell array represents the initial conditions within one channel.

   The output of the delay block is

   $$\begin{bmatrix}11 & 12\\13 & 14\end{bmatrix}, \begin{bmatrix}15 & 16\\17 & 18\end{bmatrix}, \begin{bmatrix}1 & 1\\1 & 1\end{bmatrix}, \begin{bmatrix}2 & 2\\2 & 2\end{bmatrix}, \ldots$$

   Note how each element of the cell array represents the initial conditions within a channel. The first element, a vector, represents the initial conditions within channel 1. The second element, a vector, represents the initial conditions within channel 2, and so on. Different initial conditions are used for each channel and within the channels.

Frame-Based Operation

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.

If the input is frame based, the **Delay** parameter can be a scalar integer by which to equally delay all channels or a vector whose length is equal to the number of channels.

There are four different choices for initial conditions. The initial conditions can be the same or different for each channel. They can also be constant or varying along each channel. The next sections describe the behavior of the block for each of these four cases:

- "Case 1 — Use the Same Initial Conditions for Each Channel and Within a Channel" on page 7-168
- "Case 2 — Use Different Initial Conditions for Each Channel and the Same Initial Conditions Within a Channel" on page 7-169
- "Case 3 — Use the Same Initial Conditions for Each Channel and Different Initial Conditions Within a Channel" on page 7-170
- "Case 4— Use Different Initial Conditions for Each Channel and Within a Channel" on page 7-171

Case 1 — Use the Same Initial Conditions for Each Channel and Within a Channel

Enter a scalar value for the initial conditions. This value is used as the constant initial condition value for each of the channels.

For example, suppose your input is a frame-based matrix.

$$\begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 & 4 \\ 5 & 5 & 5 \\ 6 & 6 & 6 \end{bmatrix}, \begin{bmatrix} 7 & 7 & 7 \\ 8 & 8 & 8 \\ 9 & 9 & 9 \end{bmatrix}, \cdots$$

You want the initial conditions of your three-channel signal to be identical and zero for the first frame:

1 For the **Delay (frames)** parameter, type 1.

2 Clear the **Specify different initial conditions for each channel** and the **Specify different initial conditions within a channel** check boxes.

3 For the **Initial conditions** parameter, specify a scalar value of 0.

The output of the delay block is

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 & 4 \\ 5 & 5 & 5 \\ 6 & 6 & 6 \end{bmatrix}, \begin{bmatrix} 7 & 7 & 7 \\ 8 & 8 & 8 \\ 9 & 9 & 9 \end{bmatrix}, \cdots$$

Note how 0, the scalar initial condition value, is used across the channels and within the channels for the first frame. This frame is the output at sample time zero.

Case 2 — Use Different Initial Conditions for Each Channel and the Same Initial Conditions Within a Channel

The initial conditions must be a vector of length N, where $N \geq 1$. N is also equal to the number of channels in your signal. These initial condition values are used as the constant initial condition value for each of the channels.

For example, suppose your input is a frame-based matrix.

$$\begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 & 4 \\ 5 & 5 & 5 \\ 6 & 6 & 6 \end{bmatrix}, \begin{bmatrix} 7 & 7 & 7 \\ 8 & 8 & 8 \\ 9 & 9 & 9 \end{bmatrix}, \cdots$$

You want the initial conditions of your three-channel signal to be `[0 10 20]` for the first frame:

1 For the **Delay (frames)** parameter, type 1.

2 Select the **Specify different initial conditions for each channel** check box.

3 Clear the **Specify different initial conditions within a channel** check box.

4 For the **Initial conditions** parameter, type `[0 10 20]`.

The output of the delay block is

$$\begin{bmatrix} 0 & 10 & 20 \\ 0 & 10 & 20 \\ 0 & 10 & 20 \end{bmatrix}, \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 & 4 \\ 5 & 5 & 5 \\ 6 & 6 & 6 \end{bmatrix}, \begin{bmatrix} 7 & 7 & 7 \\ 8 & 8 & 8 \\ 9 & 9 & 9 \end{bmatrix}, \cdots$$

Note how the initial condition vector is expanded to create the frame that is output at sample time zero. Different initial conditions are used for each channel, but the same initial condition value is used with a channel.

Case 3 — Use the Same Initial Conditions for Each Channel and Different Initial Conditions Within a Channel

In this case, the **Delay** parameter can be a scalar integer by which to equally delay all channels or a vector whose length is equal to the number of channels. All the values of this vector must be equal.

Enter the initial conditions as a vector. These values are used as the initial condition value along each of the channels to be delayed. The initial condition vector must have length equal to the value of the **Delay (frames)** parameter multiplied by the frame length. For example, if you want to delay your signal by two frames with frame length two and an initial condition value of 3, enter your initial condition vector as `[3 3 3 3]`.

For example, suppose your input is a frame-based matrix.

$$\begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 & 4 \\ 5 & 5 & 5 \\ 6 & 6 & 6 \end{bmatrix}, \begin{bmatrix} 7 & 7 & 7 \\ 8 & 8 & 8 \\ 9 & 9 & 9 \end{bmatrix}, \cdots$$

You want the initial conditions of your three-channel signal to be the same along each of the channels to be delayed:

1 For the **Delay (frame)** parameter, type `1`.

2 Clear the **Specify different initial conditions for each channel** check box.

3 Select the **Specify different initial conditions within a channel** check box.

4 For the **Initial conditions** parameter, type `[10 20 30]`.

The output of the delay block is

$$\begin{bmatrix} 10 & 10 & 10 \\ 20 & 20 & 20 \\ 30 & 30 & 30 \end{bmatrix}, \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 & 4 \\ 5 & 5 & 5 \\ 6 & 6 & 6 \end{bmatrix}, \begin{bmatrix} 7 & 7 & 7 \\ 8 & 8 & 8 \\ 9 & 9 & 9 \end{bmatrix}, \cdots$$

Note how the initial condition vector defines the initial condition values within each of the three channels. The same initial conditions are used for each channel, but different initial condition values are used with a channel.

Case 4— Use Different Initial Conditions for Each Channel and Within a Channel

Enter a cell array for your initial condition values. Or, if you have a scalar delay value, you can enter the initial conditions as a matrix.

For example, suppose your input is a frame-based matrix.

$$\begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 & 4 \\ 5 & 5 & 5 \\ 6 & 6 & 6 \end{bmatrix}, \begin{bmatrix} 7 & 7 & 7 \\ 8 & 8 & 8 \\ 9 & 9 & 9 \end{bmatrix}, \cdots$$

You want the initial conditions of your three-channel signal to be different for each channel and along each channel.

1 For the **Delay (frames)** parameter, type `1`.

2 Select the **Specify different initial conditions for each channel** and the **Specify different initial conditions within a channel** check boxes.

3 For the **Initial conditions** parameter, type either `[10 20 30; 40 50 60; 70 80 90]` or `{[10 40 70];[20 50 80];[30 60 90]}`. Note that each cell of the cell array represents the delay along one channel.

Regardless of whether you use a matrix or cell array, the output of the delay block is

$$\begin{bmatrix} 10 & 20 & 30 \\ 40 & 50 & 60 \\ 70 & 80 & 90 \end{bmatrix}, \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 & 4 \\ 5 & 5 & 5 \\ 6 & 6 & 6 \end{bmatrix}, \begin{bmatrix} 7 & 7 & 7 \\ 8 & 8 & 8 \\ 9 & 9 & 9 \end{bmatrix} \cdots$$

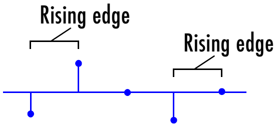
Note how the initial condition matrix is the output at sample time zero. The elements of the initial condition cell array define the initial condition values within each channel. The first element, a vector, represents the initial conditions within channel 1. The second element, a vector, represents the initial conditions within channel 2, and so on. Different initial conditions are used for each channel and within the channels.

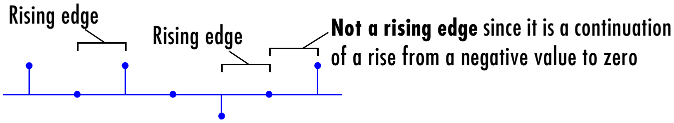
Resetting the Delay

The Delay block resets the delay whenever it detects a reset event at the optional `Rst` port. The reset signal rate must be a positive integer multiple of the rate of the data signal input.

The reset event is specified by the **Reset port** parameter, and can be one of the following:

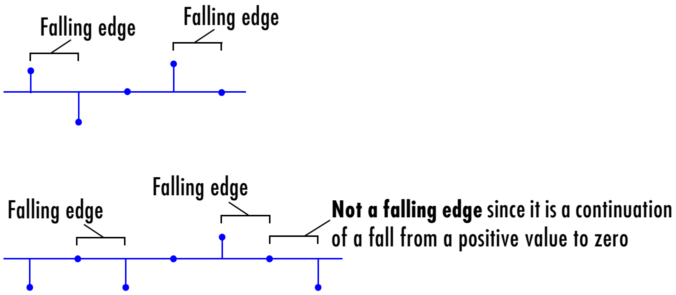
- `None` disables the `Rst` port.
- `Rising edge` triggers a reset operation when the `Rst` input does one of the following:
  - Rises from a negative value to a positive value or zero
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)




Rising edge
Rising edge
Not a rising edge since it is a continuation of a rise from a negative value to zero

- `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

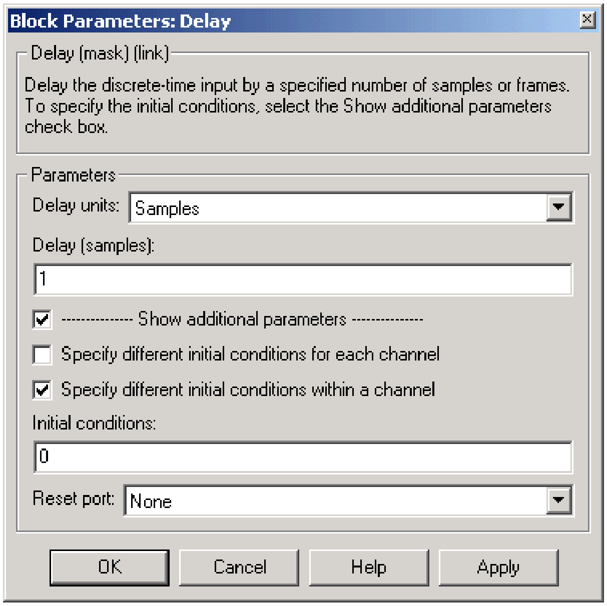
- Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero (see the following figure)


Falling edge
Falling edge
Falling edge
Falling edge
Not a falling edge since it is a continuation
of a fall from a positive value to zero

- `Either edge` triggers a reset operation when the `Rst` input is `Rising edge` or `Falling edge` (as described above).
- `Non-zero sample` triggers a reset operation at each sample time that the `Rst` input is not zero.

Note When running simulations in the Simulink `MultiTasking` mode, sample-based reset signals have a one-sample latency, and frame-based reset signals have one frame of latency. Thus, there is a one-sample or one-frame delay between the time the block detects a reset event, and when it applies the reset. For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" on page 2-98.

Dialog Box



Block Parameters: Delay

Delay (mask) (link)

Delay the discrete-time input by a specified number of samples or frames. To specify the initial conditions, select the Show additional parameters check box.

Parameters

Delay units: Samples

Delay (samples): 1

☑ -------------- Show additional parameters --------------

□ Specify different initial conditions for each channel

☑ Specify different initial conditions within a channel

Initial conditions: 0

Reset port: None

Delay units

Select whether you want to delay your input by a specified number of `Samples` or `Frames`. You can choose to delay your signal by a certain number of samples or frames regardless of whether your input is sample or frame based.

Delay (samples) or Delay (frames)

See “Sample-Based Operation” on page 7-163 and “Frame-Based Operation” on page 7-168 for a description of what format to use for each configuration of the block dialog.

Specify different initial conditions for each channel

Select this check box if you want the initial conditions to vary across the channels. If this check box is not selected, the initial conditions are the same across the channels.

Specify different initial conditions within a channel

Select this check box if you want the initial conditions to vary within the channels. If this check box is not selected, the initial conditions are the same within the channels.

Initial conditions

Enter a scalar, vector, matrix, or cell array of initial condition values depending on your choice for the **Specify different initial conditions for each channel** and **Specify different initial conditions within a channel** check boxes. See "Sample-Based Operation" on page 7-163 and "Frame-Based Operation" on page 7-168 for a description of what format to use for each configuration of the block dialog.

Reset port

Determines the reset event that causes the block to reset the delay. For more information, see "Resetting the Delay" on page 7-172.

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

See Also

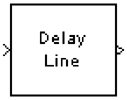
Unit Delay	Simulink
Variable Fractional Delay	DSP Blockset
Variable Integer Delay	DSP Blockset

Delay Line

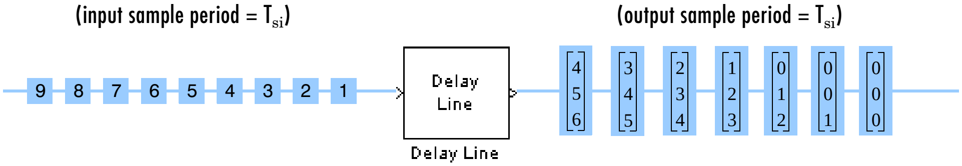
Purpose Rebuffer a sequence of inputs with a one-sample shift

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.


(input sample period = T_{si})
(output sample period = T_{si})
9
8
7
6
5
4
3
2
1
Delay
Line
Delay Line
4 5 6
3 4 5
2 3 4
1 2 3
0 1 2
0 0 1
0 0 0

Note that the first output of the block in the example above is all zeros; this is because the **Initial Conditions** parameter is set to zero. Due to the latency of the Delay Line block, all outputs are delayed by one frame, the entries of which are defined by the **Initial Conditions** parameter.

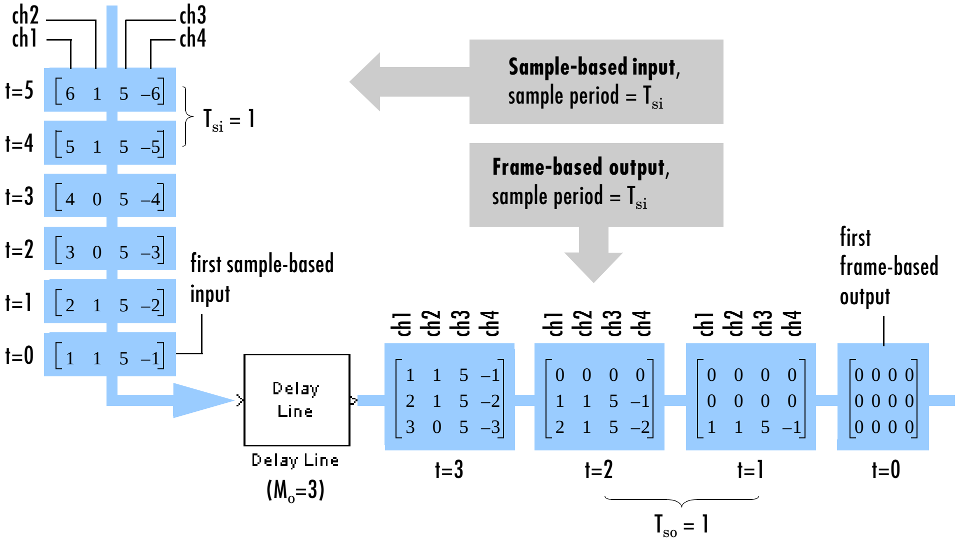
Sample-Based Operation

In sample-based operation, the Delay Line block buffers a sequence of sample-based length-N vector inputs (1-D, row, or column) into a sequence of overlapping frame-based M_o-by-N matrix outputs, where M_o is specified by the **Delay line size** parameter (M_o>1). That is, each input vector becomes a *row* in the frame-based output matrix.

At each sample time the new input vector is added in the last row of the output, so each output overlaps the previous output by M_o-1 samples. Therefore, the output sample period and frame period is the same as the input sample period (T_{so}=T_{si}, and T_{fo}=T_{si}). When M_o=1, the input is simply passed through to the output and retains the same dimension, but becomes frame based. The latency of the block always causes an initial delay in the output; the value of the first output is specified by the **Initial conditions** parameter (see “Initial Conditions” on page 7-178 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” on page 7-178 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

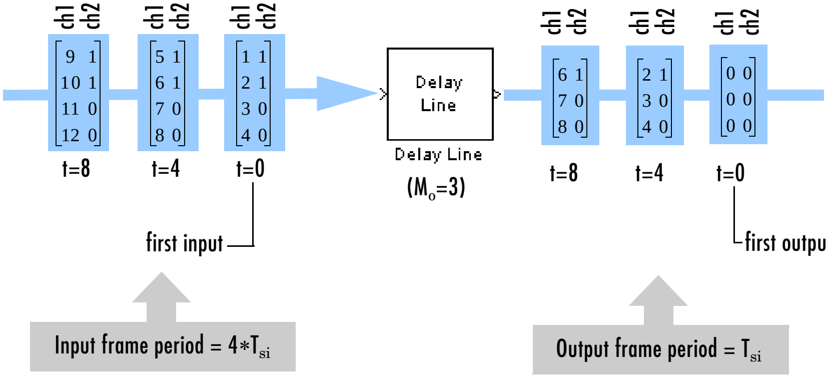
In frame-based operation, the Delay Line block rebuffers a sequence of frame-based M_i-by-N matrix inputs into a sequence of frame-based M_o-by-N matrix outputs, where M_o is the output frame size specified by the **Delay line size** parameter. Depending on whether M_o is greater than, less than, or 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.

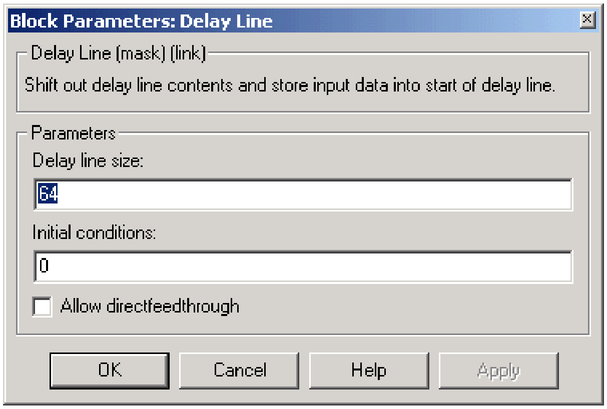


The first output frame in the example is a product of the latency of the Delay Line block; it is all zeros because the **Initial conditions** is set to be zero. Since the input frame size, 4, is larger than the output frame size, 3, only the last three samples in each input frame are propagated to the 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



Block Parameters: Delay Line

Delay Line (mask) (link)

Shift out delay line contents and store input data into start of delay line.

Parameters

Delay line size:

64

Initial conditions:

0

☐ Allow directfeedthrough

Delay line size

The number of rows in output matrix, M_o.

Initial conditions

The value of the block's initial output, a scalar, vector, or matrix.

Allow direct feedthrough

If you select this check box, the input data is not delayed by an extra frame before it is available at the output buffer. Instead, the input data is available immediately at the output port of the block.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point
- Custom data types
- Boolean
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

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

See Also

Buffer	DSP Blockset
Triggered Delay Line	DSP Blockset

See “Buffering Sample-Based and Frame-Based Signals” on page 2-47 for related information.

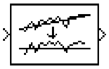
Purpose

Remove a linear trend from a vector

Library

Statistics

Description



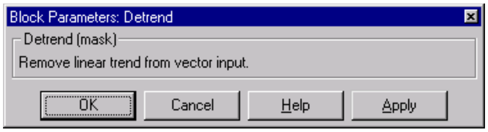
The Detrend block removes a linear trend from the length-M input vector, u, by subtracting the straight line that best fits the data in the least squares sense.

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



Block Parameters: Detrend

Detrend (mask)

Remove linear trend from vector input.

Supported Data Types

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

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

See Also

Cumulative Sum	DSP Blockset
Difference	DSP Blockset
Least Squares Polynomial Fit	DSP Blockset
Unwrap	DSP Blockset
`detrend`	MATLAB

Difference

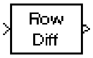
Purpose

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 *j*th 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 *j*th 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.

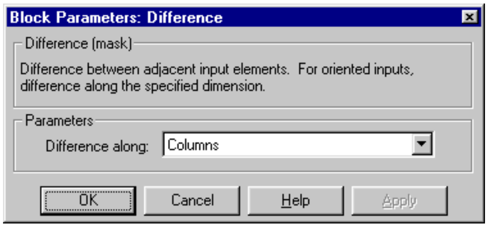
```
y = diff(u,[],2)         % Equivalent MATLAB code
```

The output is an M-by-(N-1) matrix whose *i*th 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



Block Parameters: Difference

Difference (mask)

Difference between adjacent input elements. For oriented inputs, difference along the specified dimension.

Parameters

Difference along: Columns

Difference along

The dimension along which to compute element-to-element differences. `Columns` specifies columnwise differencing, while `Rows` specifies rowwise differencing. Nontunable.

Supported Data Types

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

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

See Also

Cumulative Sum	DSP Blockset
`diff`	MATLAB

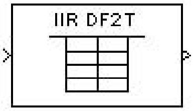
Purpose

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 and fixed-point filters for which you know the coefficients. The following DSP Blockset blocks also implement digital filters, but serve slightly different purposes:

- Digital Filter Design — Use to design, analyze, and then efficiently implement floating-point filters. This block provides the same exact filter implementation as the Digital Filter block for floating-point signals.
- Filter Realization Wizard — Use to implement floating-point or fixed-point filters built from Sum, Gain, and Unit Delay blocks. You can either design the filter using block filter design and analysis parameters, or import the coefficients of a filter that you designed elsewhere.

The Digital Filter block independently filters each channel of the input signal with a specified digital IIR or FIR filter. The block can implement *static filters* with fixed coefficients, as well as *time-varying filters* with coefficients that change over time. You can tune the coefficients of a static filter during simulation.

This block filters each channel of the input signal independently over time. The output size, frame status, and dimension 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 this block numerically match the outputs of the Digital Filter Design block and of the `dfilt` function in the Signal Processing Toolbox.

Sections of This Reference Page

- “Supported Filter Structures” on page 7-185
- “Specifying Static Filters” on page 7-186
- “Specifying Time-Varying Filters” on page 7-188

- "Specifying the SOS Matrix (Biquadratic Filter Coefficients)" on page 7-192
- "Specifying Initial Conditions" on page 7-193
- "Fixed-Point Data Types" on page 7-197
- "Examples" on page 7-200
- "Dialog Box" on page 7-201
- "Supported Data Types" on page 7-211
- "See Also" on page 7-211

Supported Filter Structures

The selection of filter structures offered in the **Filter structure** parameter depends on whether you set the **Transfer function type** to `IIR (poles & zeros)`, `IIR (all poles)`, or `FIR (all zeros)`, as summarized in the table below.

Note The IIR Biquadratic Direct Form II Transposed and FIR Direct Form modes currently support fixed-point signals as well as floating-point signals.

The table also shows the vector or matrix of filter coefficients you must provide for each filter structure. For more information on how to specify filter coefficients for various filter structures, see "Specifying Static Filters" on page 7-186 and "Specifying Time-Varying Filters" on page 7-188.

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 (SOS)[1]	M-by-6 second-order section (SOS) matrix. See "Specifying the SOS Matrix (Biquadratic Filter Coefficients)" on page 7-192.
`IIR (all poles)`	**Direct form** **Direct form transposed**	Denominator coefficients vector [a0, a1, a2, ..., am]
	Lattice AR	Reflection coefficients vector [k1, k2, ..., kn]
`FIR (all zeros)`	**Direct form**[2] **Transposed direct form**	Numerator coefficients vector [b0, b1, b2, ..., bn]
	Lattice MA	Reflection coefficients vector [k1, k2, ..., kn]

[1] Supported for fixed-point as well as floating-point signals.

[2] Supported for fixed-point as well as floating-point signals.

Specifying Static Filters

To specify a *static filter* whose coefficients are fixed in time, set the **Coefficient source** parameter to `Specify via dialog`. Depending on the filter structure, you need to enter your filter coefficients into one or more of the following parameters. The block disables all the irrelevant parameters. To see which of these parameters correspond to each filter structure, see the Filter Structures and Filter Coefficients table above:

- **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 7-192.

Tuning the Filter Coefficient Values During Simulation. To change the static filter coefficients during simulation, double-click the block, type in the new vector(s) of filter coefficients, and click **OK**. You cannot change the filter order, so you cannot change the number of elements in the vector(s) of filter coefficients.

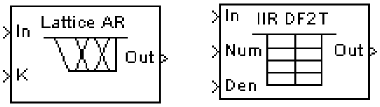
Specifying Time-Varying Filters

Note This block does not support time-varying Biquadratic 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.



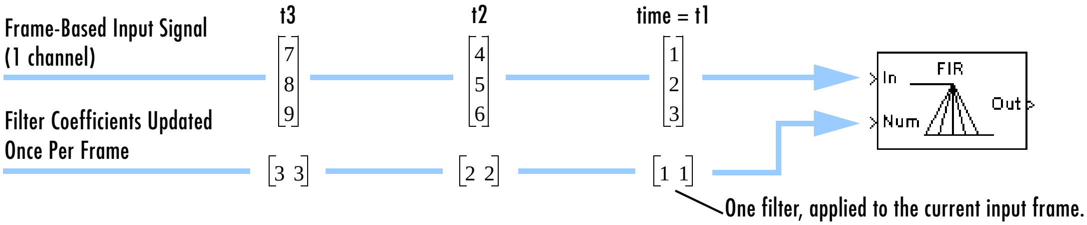
2. Set the **Coefficient update rate** parameter to `One filter per frame` or `One filter per sample` depending on how often you want to update the filter coefficients. To learn more, see “Setting the Coefficient Update Rate” on page 7-189.
3. Provide vectors of numerator, denominator, or reflection coefficients to the block input ports for filter coefficients. The series of vectors *must arrive at their ports at a specific rate*, and *must be of certain lengths*. To learn more, see “Providing Filter Coefficient Vectors at Block Input Ports” on page 7-190.
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 7-192.

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

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

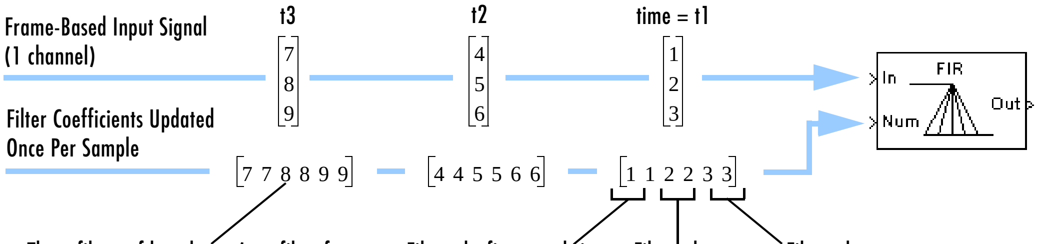
The following figure shows the block filtering one channel; however, the block can filter multiple channels. Note that the block *can* apply a single filter to multiple channels, but *cannot* apply a different filter to each channel.

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



One filter, applied to the current input frame.

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



Three filters of length two (one filter for each sample in the current input frame).

Filters the first sample in the current input frame.

Filters the second sample.

Filters the third sample.

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

- Length requirements — See the table below called Length Requirements for Time-Varying Filter Coefficient Vectors
- Rate requirements — See the table called “Rate Requirements for Time-Varying Filter Coefficient Vectors” on page 7-191.

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

Length Requirements for Time-Varying Filter Coefficient Vectors

Coefficient Update Rate	**How to Specify Filter Coefficient Vectors** (Also see the previous figure)	**Length Requirements**
Once per frame	Each coefficient vector corresponds to one input frame, and represents one filter. Specify each vector as you would any static filter: [b0, b1, b2, ..., bn], [a0, a1, a2, ..., am], or [k1, k2, ..., kn]	None
Once per sample	Each coefficient vector corresponds to one input frame, 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.)

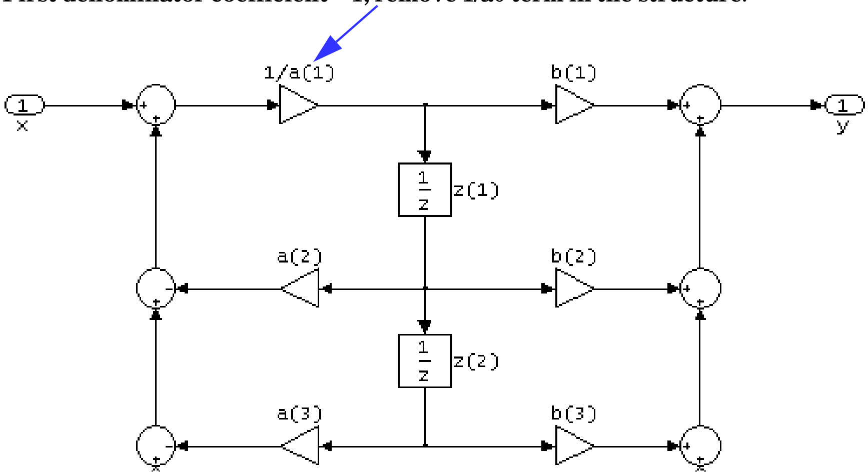
The time-varying filter coefficient vectors can be sample- or frame-based row or column vectors. The vector of filter coefficients must arrive at their input ports at the same times that the frames of input data arrive at their input port, as indicated in the following table.

Rate Requirements for Time-Varying Filter Coefficient Vectors

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

Removing the 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 / a0` 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
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.

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

Note The block uses a value of 1 for the zero-delay denominator coefficients (a_{11} to $a_{1\mathrm{M}}$) 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 the following table on Valid Initial Conditions and “Number of Delay Elements (Filter States)” on page 7-195. The **Initial conditions** parameter may take one of four forms as described in the following table.

Valid Initial Conditions

Initial Condition	Examples	Description
Scalar	5 Each delay element for each channel is set to 5.	The block initializes all delay elements in the filter to the scalar value.
Vector (for applying the same delay elements to each channel)	For a filter with two delay elements: [d_1 d_2] The delay elements for all channels are d1 and d2.	Each vector element specifies a unique initial condition for a corresponding delay element. The block applies the same vector of initial conditions to each channel of the input signal. The vector length must equal the number of delay elements in the filter (specified in the Number of Delay Elements (Filter States) table).

Valid Initial Conditions (Continued)

Initial Condition	Examples	Description
Vector or matrix (for applying different delay elements to each channel)	For a 3-channel input signal and a filter with two delay elements: $[\mathrm{d}_1\ \mathrm{d}_2\ \mathrm{D}_1\ \mathrm{D}_2\ d_1\ d_2]$ 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 the Number of Delay Elements (Filter States) table). • The matrix must have the same number of rows as the number of delay elements in the filter (specified in the Number of Delay Elements (Filter States) table), and must have one column for each channel of the input signal.
Empty matrix	`[]` Each delay element for each channel is set to `0`.	The empty matrix, `[]`, is equivalent to setting the **Initial conditions** parameter to the scalar value `0`.

The number of delay elements (filter states) per input channel depends on the filter structure, as indicated in the following table.

Number of Delay Elements (Filter States)

Filter Structure	Number of Delay Elements Per Channel
Direct form	`#_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`

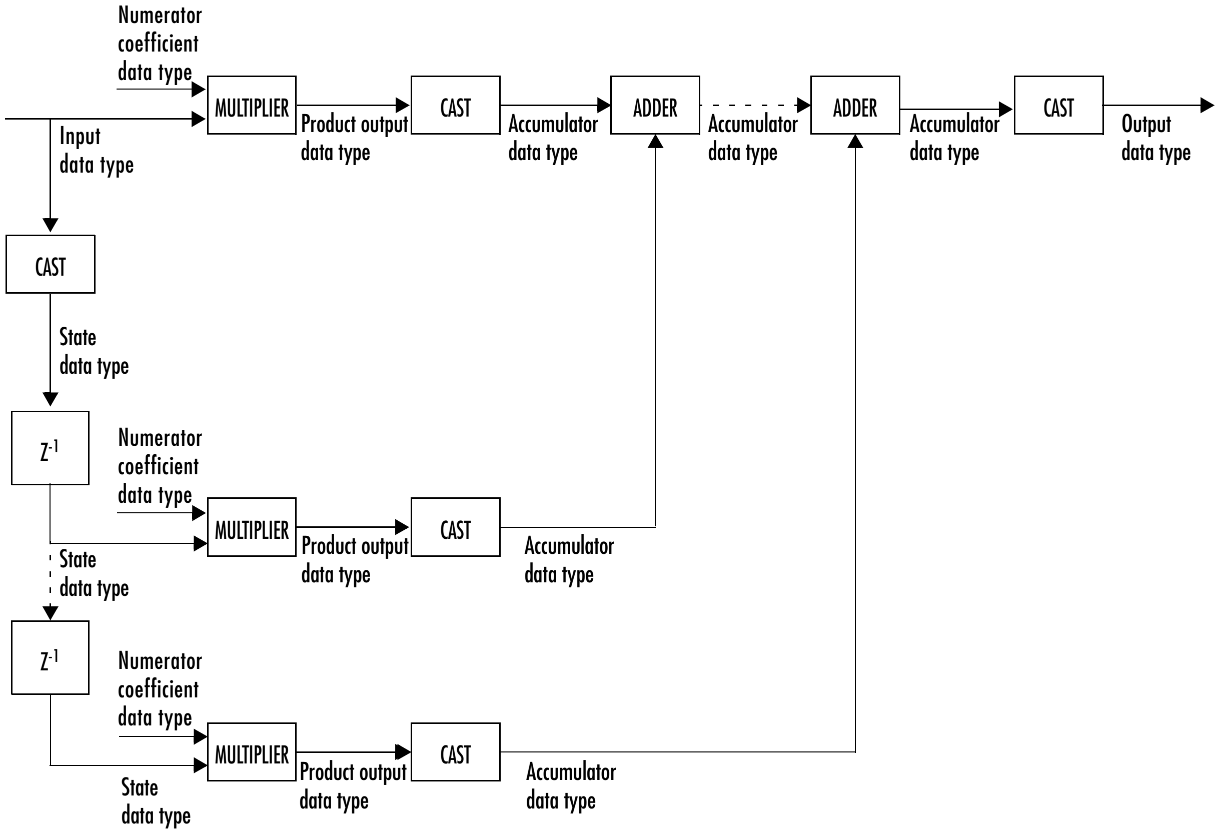
Number of Delay Elements (Filter States) (Continued)

Filter Structure	Number of Delay Elements Per Channel
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`

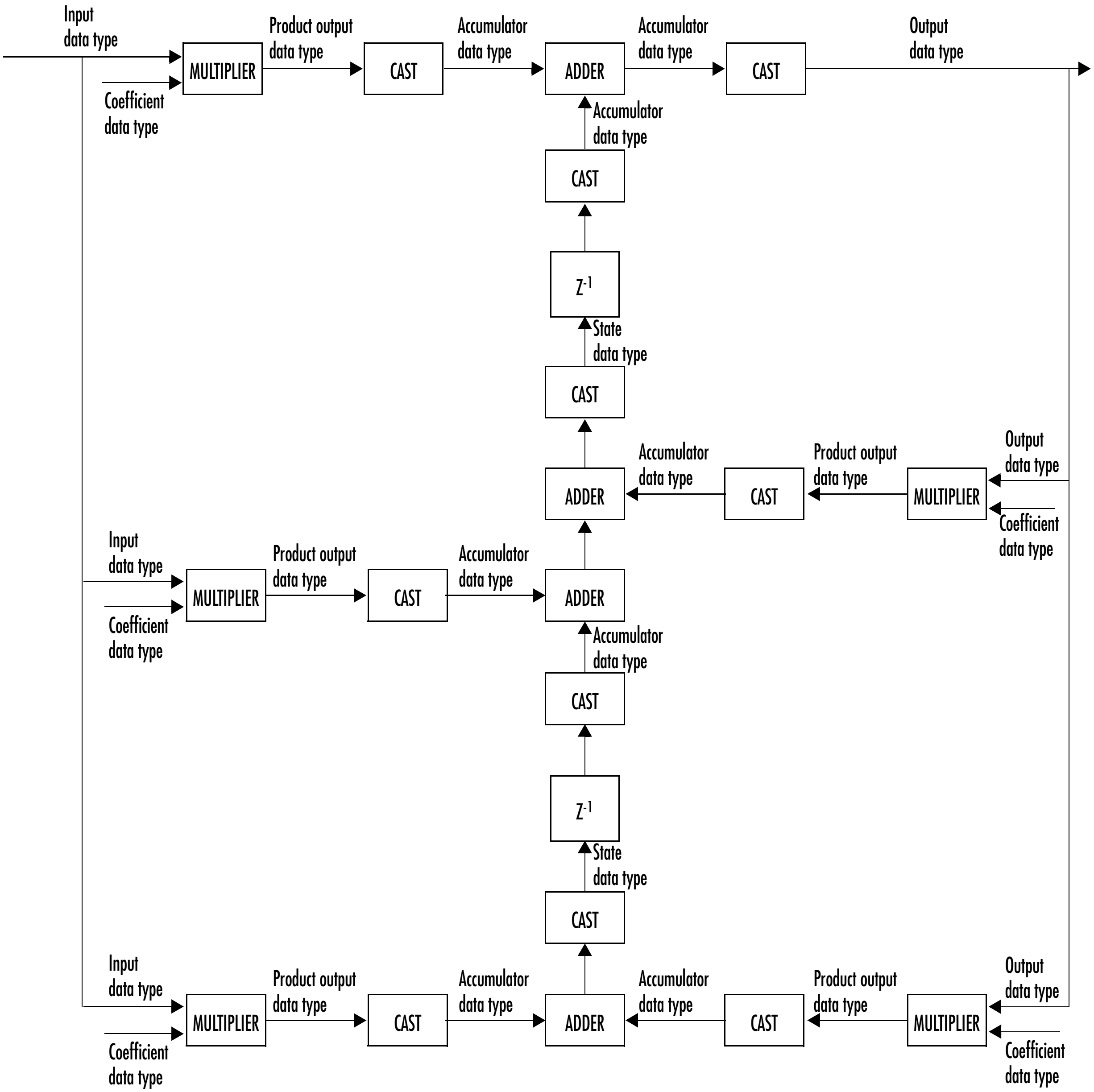
Fixed-Point Data Types

The diagrams in this section show the data types used for the filter structures that are currently supported for fixed-point signals. You can set the coefficient, output, accumulator, product output, and state data types shown in these diagrams in the block mask. This is discussed in “Dialog Box” on page 7-201.

The following diagram shows the data types used for the FIR Direct Form filter for fixed-point signals. A filter with two stages is shown; additional stages can be inserted at the dashed lines.

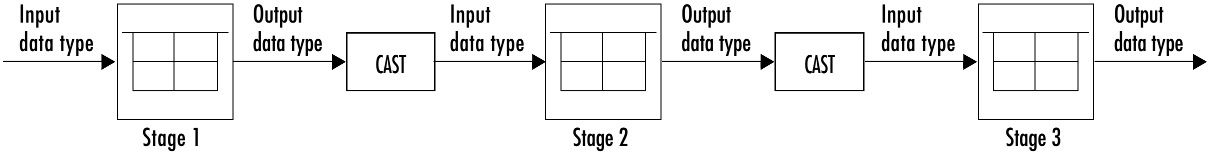


The following diagram shows the data types used for one section of the IIR Biquadratic Direct Form II Transposed filter for fixed-point signals.


Input
data type
Coefficient
data type
MULTIPLIER
Product output
data type
CAST
Accumulator
data type
ADDER
Accumulator
data type
CAST
Output
data type
Accumulator
data type
CAST
Z-1
State
data type
CAST
Accumulator
data type
Product output
data type
Output
data type
ADDER
CAST
MULTIPLIER
Coefficient
data type
Input
data type
MULTIPLIER
Product output
data type
CAST
Accumulator
data type
ADDER
Coefficient
data type
Accumulator
data type
CAST
Z-1
State
data type
CAST
Input
data type
MULTIPLIER
Product output
data type
CAST
Accumulator
data type
ADDER
Accumulator
data type
CAST
Product output
data type
MULTIPLIER
Output
data type
Coefficient
data type
Coefficient
data type

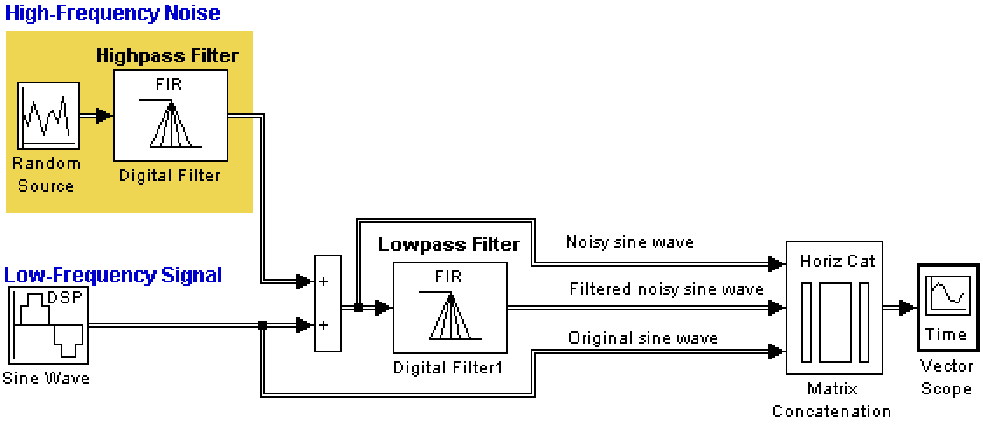
Note For this filter, the numerator coefficient word and fraction lengths must be the same as the denominator coefficient word and fraction lengths.

For a multiple-stage IIR Biquadratic Direct Form II Transposed filter, the output of each stage is cast to the input data type before entering the next stage of the filter. The following diagram shows an example with three stages.

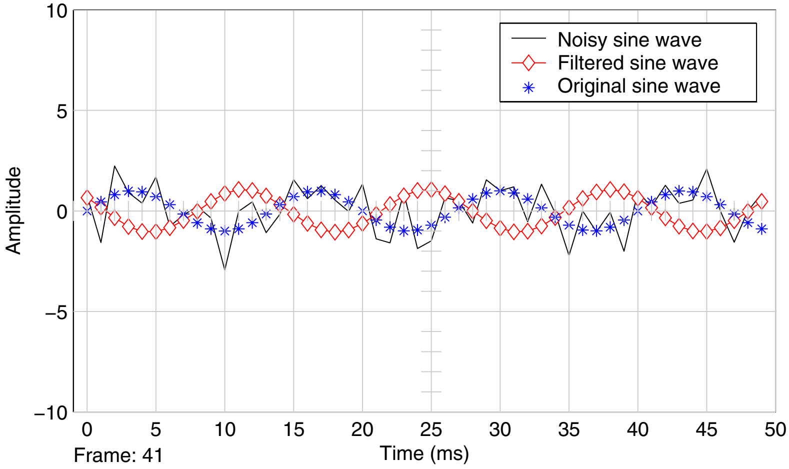


Examples

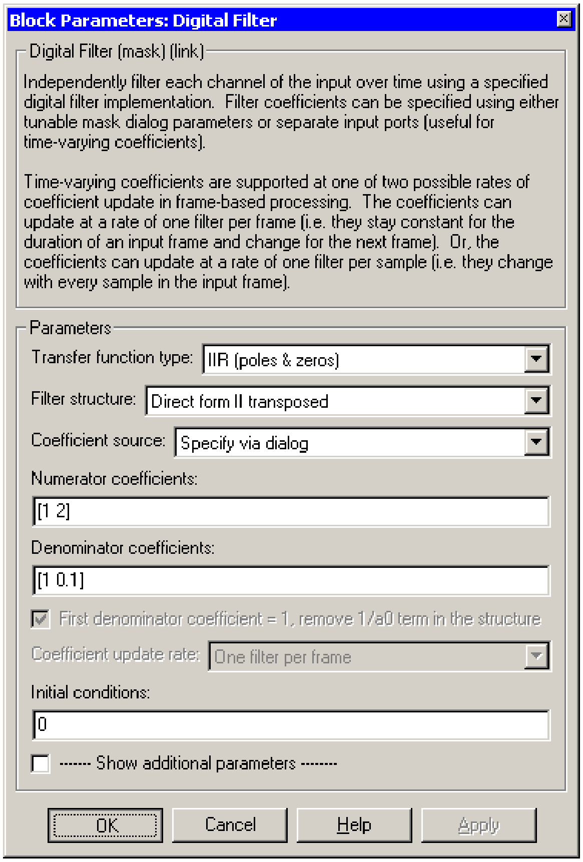
The following Simulink model is `digitalfilter_tut.mdl`. To build the model yourself, follow the step-by-step instructions in “Digital Filter Block” on page 3-2, which also provides a full explanation of the model.



After building and running the model, the vector scope display should look like the following.



Dialog Box



Block Parameters: Digital Filter

Digital Filter (mask) (link)

Independently filter each channel of the input over time using a specified digital filter implementation. Filter coefficients can be specified using either tunable mask dialog parameters or separate input ports (useful for time-varying coefficients).

Time-varying coefficients are supported at one of two possible rates of coefficient update in frame-based processing. The coefficients can update at a rate of one filter per frame (i.e. they stay constant for the duration of an input frame and change for the next frame). Or, the coefficients can update at a rate of one filter per sample (i.e. they change with every sample in the input frame).

Parameters

Transfer function type: IIR (poles & zeros)

Filter structure: Direct form II transposed

Coefficient source: Specify via dialog

Numerator coefficients:

[1 2]

Denominator coefficients:

[1 0.1]

☑ First denominator coefficient = 1, remove 1/a0 term in the structure

Coefficient update rate: One filter per frame

Initial conditions:

0

☐ ------ Show additional parameters -------

Transfer function type

Select the type of transfer function of the filter; `IIR (poles & zeros)`, `IIR (all poles)`, or `FIR (all zeros)`. Refer to “Supported Filter Structures” on page 7-185 for more information.

Filter structure

Select the filter structure. The selection of available structures varies depending the setting of the **Transfer function type** parameter. Refer to “Supported Filter Structures” on page 7-185 for more information.

Coefficient source

Choose how you will specify filter coefficients: via dialog parameters or through input ports. If you select `Specify via dialog`, the applicable coefficients parameters are available. If you select `Input ports(s)`, filter coefficients must come in through block ports.

Numerator coefficients

Specify the vector of numerator coefficients of the filter's transfer function. This parameter is only visible when the **Coefficient source** parameter is set to `Dialog parameter(s)` *and* when the selected filter structure lends itself to specification with numerator coefficients. Tunable.

Denominator coefficients

Specify the vector of denominator coefficients of the filter's transfer function. This parameter is only visible when the **Coefficient source** parameter is set to `Dialog parameter(s)` *and* when the selected filter structure lends itself to specification with denominator coefficients. Tunable.

Reflection coefficients

(Not shown in dialog above). Specify the vector of reflection coefficients of the filter's transfer function. This parameter is only visible when the **Coefficient source** parameter is set to `Dialog parameter(s)` *and* when the selected filter structure lends itself to specification with reflection coefficients. Tunable.

SOS matrix (Mx6)

(Not shown in dialog above). Specify an M-by-6 *SOS matrix* containing coefficients of a second-order section (SOS) filter, where M is the number of sections. You can use the `ss2sos` and `tf2sos` functions from the Signal Processing Toolbox to check whether your SOS matrix is valid. For more on the requirements of the SOS matrix, see "Specifying the SOS Matrix (Biquadratic Filter Coefficients)" on page 7-192. This parameter is only visible when the **Coefficient source** parameter is set to `Dialog parameter(s)` *and* when the selected filter structure lends itself to specification with an SOS matrix. Tunable.

First denominator coefficient = 1, remove 1/a0 term in the structure

Select this parameter to reduce the number of computations the block must make to produce the output by omitting the $1 / a_0$ term in the filter structure. The block output is invalid if you select this parameter when the first denominator filter coefficient is *not* always 1 for your time-varying filter. This parameter is only enabled when the **Coefficient source** parameter is set to `Input port(s)` *and* when the selected filter structure lends itself to this specification. See “Removing the 1/a0 Term in the Filter Structure” on page 7-192 for a diagram and details.

Coefficient update rate

Specify how often the block updates time-varying filters; once per sample or once per frame. This parameter only affects the output when the input signal is frame based. This parameter is only enabled when the **Coefficient source** parameter is set to `Input port(s)`. For more information, see “Specifying Time-Varying Filters” on page 7-188.

Initial conditions

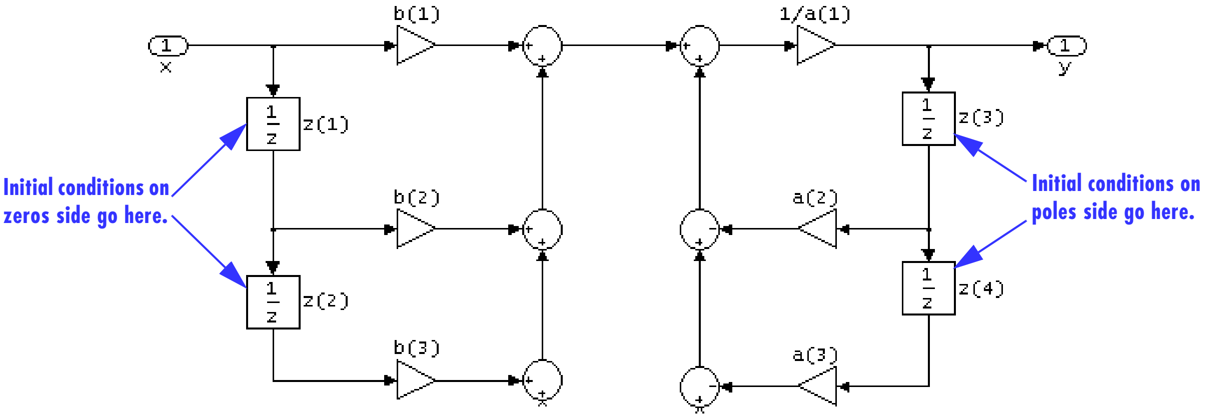
Specify the initial conditions of the filter states. To learn how to specify initial conditions, see “Specifying Initial Conditions” on page 7-193.

Initial conditions on zeros side

(Not shown in dialog above). Specify the initial conditions for the filter states on the side of the filter structure with the zeros (b_0, b_1, b_2, ...); see the diagram below. This parameter is enabled only when the filter has both poles and zeros, *and* when you select a structure such as direct form I, which has separate filter states corresponding to the poles (a_k) and zeros (b_k). To learn how to specify initial conditions, see “Specifying Initial Conditions” on page 7-193.

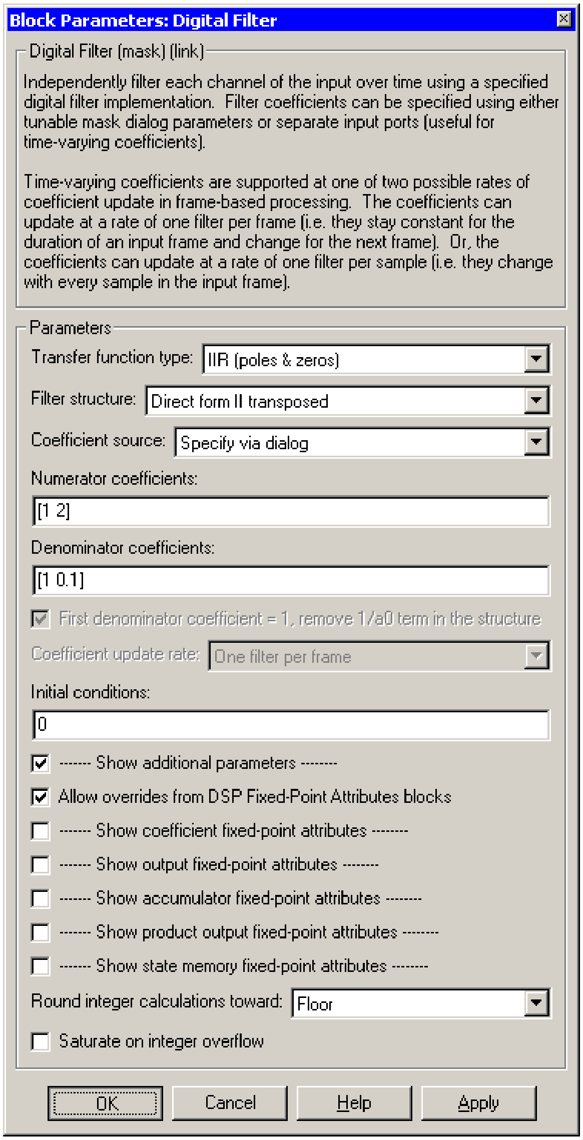
Initial conditions on poles side

(Not shown in dialog above). Specify the initial conditions for the filter states on the side of the filter structure with the poles (a_0, a_1, a_2, ...); see the diagram below. This parameter is enabled only when the filter has both poles and zeros, *and* when you select a structure such as direct form I, which has separate filter states corresponding to the poles (a_k) and zeros (b_k). To learn how to specify initial conditions, see “Specifying Initial Conditions” on page 7-193.


1
x
b(1)
1/a(1)
1
y
1
z
z(1)
1
z
z(3)
Initial conditions on zeros side go here.
Initial conditions on poles side go here.
b(2)
a(2)
1
z
z(2)
1
z
z(4)
b(3)
a(3)
+
-

Show additional parameters

If selected, additional parameters specific to implementation of the block become visible as shown.



Block Parameters: Digital Filter

Digital Filter (mask) (link)

Independently filter each channel of the input over time using a specified digital filter implementation. Filter coefficients can be specified using either tunable mask dialog parameters or separate input ports (useful for time-varying coefficients).

Time-varying coefficients are supported at one of two possible rates of coefficient update in frame-based processing. The coefficients can update at a rate of one filter per frame (i.e. they stay constant for the duration of an input frame and change for the next frame). Or, the coefficients can update at a rate of one filter per sample (i.e. they change with every sample in the input frame).

Parameters

Transfer function type: IIR (poles & zeros)

Filter structure: Direct form II transposed

Coefficient source: Specify via dialog

Numerator coefficients: [1 2]

Denominator coefficients: [1 0.1]

☑ First denominator coefficient = 1, remove 1/a0 term in the structure

Coefficient update rate: One filter per frame

Initial conditions: 0

☑ ------- Show additional parameters --------

☑ Allow overrides from DSP Fixed-Point Attributes blocks

□ ------- Show coefficient fixed-point attributes --------

□ ------- Show output fixed-point attributes --------

□ ------- Show accumulator fixed-point attributes --------

□ ------- Show product output fixed-point attributes --------

□ ------- Show state memory fixed-point attributes --------

Round integer calculations toward: Floor

□ Saturate on integer overflow

When the **Show coefficient fixed-point attributes**, **Show output fixed-point attributes**, **Show accumulator fixed-point attributes**, **Show product output fixed-point attributes**, or **Show state memory fixed-point attributes** check box is selected, additional parameters become available, as discussed in the following sections:

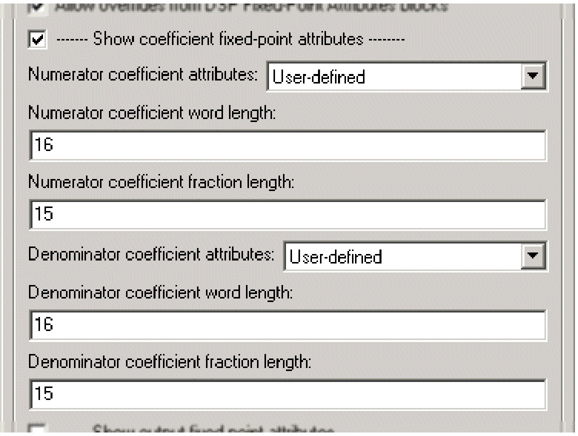
- "Show coefficient fixed-point attributes" on page 7-206
- "Show output fixed-point attributes" on page 7-207
- "Show accumulator fixed-point attributes" on page 7-208
- "Show product output fixed-point attributes" on page 7-209
- "Show state memory fixed-point attributes" on page 7-210

Allow overrides from DSP Fixed-Point Attributes blocks

If you select this parameter, fixed-point data types for this block may be set by DSP Fixed-Point Attributes blocks in your model. If this parameter is unselected, the data types are always set by the parameters in the block mask.

Show coefficient fixed-point attributes

When this parameter is selected, additional fixed-point parameters become visible as shown.



☑ Allow overrides from DSP Fixed-Point Attributes blocks

☑ ------ Show coefficient fixed-point attributes -------

Numerator coefficient attributes: User-defined

Numerator coefficient word length: 16

Numerator coefficient fraction length: 15

Denominator coefficient attributes: User-defined

Denominator coefficient word length: 16

Denominator coefficient fraction length: 15

☐ Show output fixed-point attributes

Numerator coefficient attributes

Choose how you will specify the word and fraction lengths of any numerator coefficients. Refer to "Fixed-Point Data Types" on page 7-197 for illustrations depicting the use of the numerator coefficient data type in this block.

If you select `Same as input`, the numerator coefficient word and fraction lengths are the same as those of the input to the block. If you select `User-defined`, the **Numerator coefficient word length** and **Numerator coefficient fraction length** parameters become visible.

Numerator coefficient word length

Specify the word length, in bits, of the numerator coefficients. This parameter is only visible when `User-defined` is specified for the **Numerator coefficient attributes** parameter.

Numerator coefficient fraction length

Specify the fraction length, in bits, of the numerator coefficients. This parameter is only visible when `User-defined` is specified for the **Numerator coefficients attributes** parameter.

Denominator coefficient attributes

Choose how you will specify the word and fraction lengths of any denominator coefficients. Refer to “Fixed-Point Data Types” on page 7-197 for illustrations depicting the use of the denominator coefficient data type in this block.

If you select `Same as numerator`, the denominator coefficient word and fraction lengths are the same as those of the numerator coefficients. If you select `Same as input`, they are the same as those of the input to the block. If you select `User-defined`, the **Denominator coefficient word length** and **Denominator coefficient fraction length** parameters become visible.

Denominator coefficient word length

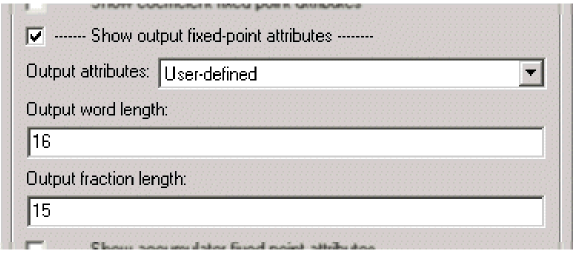
Specify the word length, in bits, of the denominator coefficients. This parameter is only visible when `User-defined` is specified for the **Denominator coefficient attributes** parameter.

Denominator coefficient fraction length

Specify the fraction length, in bits, of the denominator coefficients. This parameter is only visible when `User-defined` is specified for the **Denominator coefficients attributes** parameter.

Show output fixed-point attributes

When this parameter is selected, additional fixed-point parameters become visible as shown.



☑ ------- Show output fixed-point attributes --------

Output attributes: User-defined

Output word length: 16

Output fraction length: 15

Output attributes

Choose how you will specify the output word and fraction lengths. If you select `Same as input`, the output word and fraction lengths are the same as those of the input to the block. If you select `User-defined`, the **Output word length** and **Output fraction length** parameters become visible.

Output word length

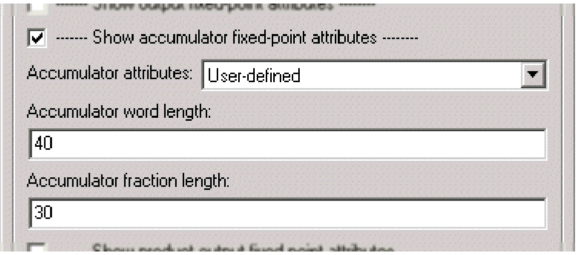
Specify the word length, in bits, of the output. This parameter is only visible when `User-defined` is specified for the **Output attributes** parameter.

Output fraction length

Specify the fraction length, in bits, of the output. This parameter is only visible when `User-defined` is specified for the **Output attributes** parameter.

Show accumulator fixed-point attributes

When this parameter is selected, additional fixed-point parameters become visible as shown.



☑ ------- Show accumulator fixed-point attributes --------

Accumulator attributes: User-defined

Accumulator word length: 40

Accumulator fraction length: 30

Accumulator attributes

Choose how you will specify the accumulator word and fraction lengths. Refer to "Fixed-Point Data Types" on page 7-197 for illustrations depicting the use of the accumulator data type in this block.

If you select `Same as output`, the accumulator word and fraction lengths are the same as those of the output of the block. If you select `User-defined`, the **Accumulator word length** and **Accumulator fraction length** parameters become visible.

Accumulator word length

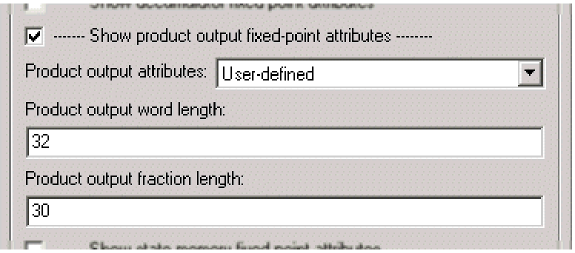
Specify the word length, in bits, of the accumulator. This parameter is only visible when `User-defined` is specified for the **Accumulator attributes** parameter.

Accumulator fraction length

Specify the fraction length, in bits, of the accumulator. This parameter is only visible when `User-defined` is specified for the **Accumulator attributes** parameter.

Show product output fixed-point attributes

When this parameter is selected, additional fixed-point parameters become visible as shown.



□ Show accumulator fixed-point attributes

☑ ------- Show product output fixed-point attributes -------

Product output attributes: User-defined

Product output word length: 32

Product output fraction length: 30

□ ------- Show state memory fixed-point attributes

Product output attributes

Choose how you will specify the product output word and fraction lengths. Refer to "Fixed-Point Data Types" on page 7-197 for illustrations depicting the use of the product output data type in this block.

If you select `Same as output`, the product output word and fraction lengths are the same as those of the output of the block. If you select `Same as` `accumulator`, they match those of the accumulator. If you select `User-defined`, the **Product output word length** and **Product output fraction length** parameters become visible.

Product output word length

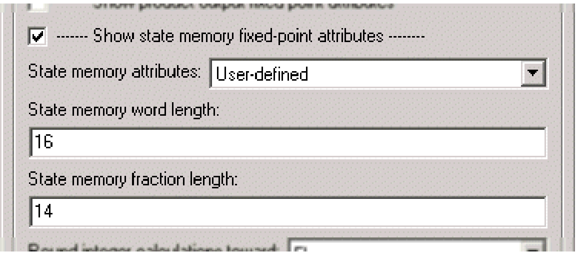
Specify the word length, in bits, of the product output. This parameter is only visible when `User-defined` is specified for the **Product output attributes** parameter.

Product output fraction length

Specify the fraction length, in bits, of the product output. This parameter is only visible when `User-defined` is specified for the **Product output attributes** parameter.

Show state memory fixed-point attributes

When this parameter is selected, additional fixed-point parameters become visible as shown.



☑ ------- Show state memory fixed-point attributes --------

State memory attributes: User-defined

State memory word length: 16

State memory fraction length: 14

State memory attributes

Choose how you will specify the state memory word and fraction lengths. Refer to “Fixed-Point Data Types” on page 7-197 for illustrations depicting the use of the state data type in this block.

If you select `Same as input`, `Same as output`, or `Same as accumulator`, the state memory word and fraction lengths are the same as those of the input, output, or accumulator of the block, respectively. If you select `User-defined`, the **State memory word length** and **State memory fraction length** parameters become visible.

State memory word length

Specify the word length, in bits, of the state memory. This parameter is only visible when `User-defined` is specified for the **State memory attributes** parameter.

State memory fraction length

Specify the fraction length, in bits, of the state memory. This parameter is only visible when `User-defined` is specified for the **State memory attributes** parameter.

Round integer calculations toward

Select the rounding mode for fixed-point operations performed by the block.

The filter coefficients do not obey this parameter; they always round to `Nearest`.

Saturate on integer overflow

If selected, overflows saturate.

The filter coefficients do not obey this parameter; they are always saturated.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point
- 8-, 16-, and 32-bit signed integers

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

See Also

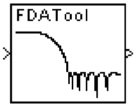
Digital Filter Design	DSP Blockset
Filter Realization Wizard	DSP Blockset
`fdatool`	Signal Processing Toolbox
`fvtool`	Signal Processing Toolbox
`sptool`	Signal Processing Toolbox

Digital Filter Design

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 the Filter Design Toolbox.

The sampling frequency, Fs, you specify in the FDATool GUI should be identical to the sampling frequency of the Digital Filter Design block's input block. If the sampling frequencies of these blocks do not match, the Digital Filter Design block returns a warning message and inherits the sampling frequency of the input block.

Sections of This Reference Page

- "Valid Inputs and Corresponding Outputs" on page 7-213
- "Designing the Filter" on page 7-213
- "Tuning the Filter During Simulation" on page 7-213
- "Examples" on page 7-214
- "Dialog Box" on page 7-215

- "Supported Data Types" on page 7-216
- "See Also" on page 7-216

Valid Inputs and Corresponding Outputs

The block accepts inputs that are sample-based or frame-based vectors and matrices. The block filters each input channel independently over time, where

- Each *column* of a frame-based vector or matrix is an independent channel.
- Each *element* of a sample-based vector or matrix is an independent channel.

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

Designing the Filter

Double-click the Digital Filter Design block to open FDATool. Use FDATool to design or import a digital FIR or IIR filter. To learn how to design filters with this block and FDATool, see the following topics:

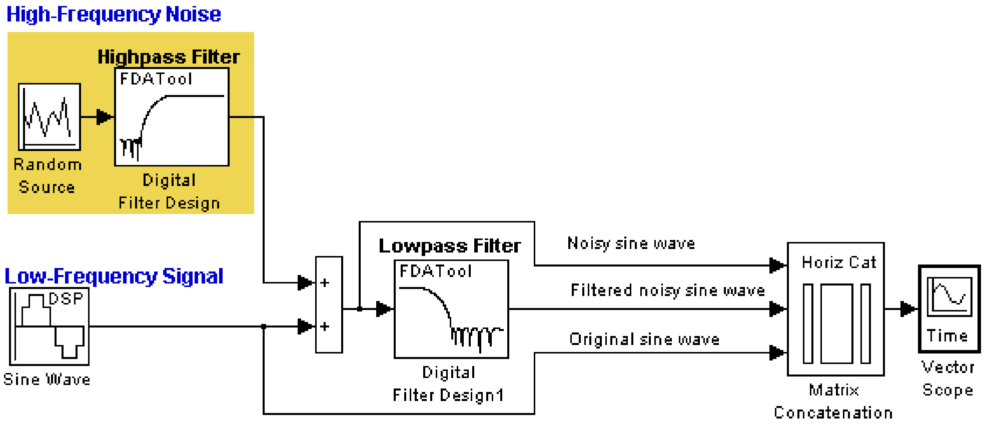
- "Digital Filter Design Block" on page 3-13
- Topic on the Filter Design and Analysis Tool (FDATool) in the Signal Processing Toolbox documentation.

Tuning the Filter During Simulation

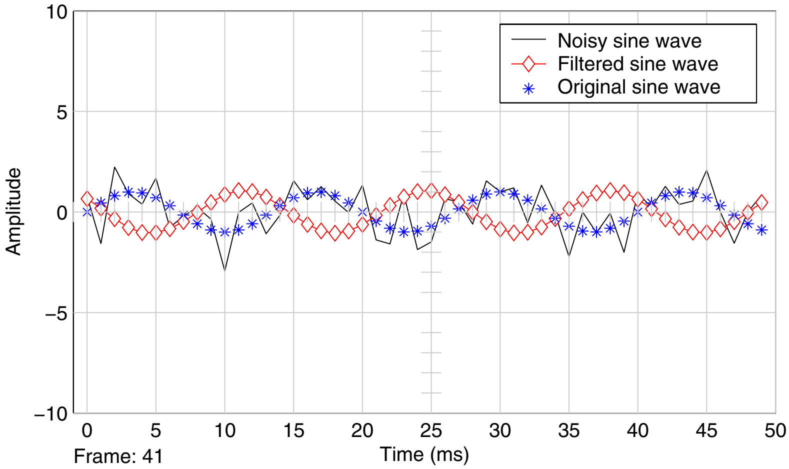
You can tune the filter specifications in FDATool during simulations as long as your changes do not modify the filter length or filter order. The block's filter updates as soon as you apply any filter changes in FDATool.

Examples

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 “Digital Filter Design Block” on page 3-13, which also provides a full explanation of the model.

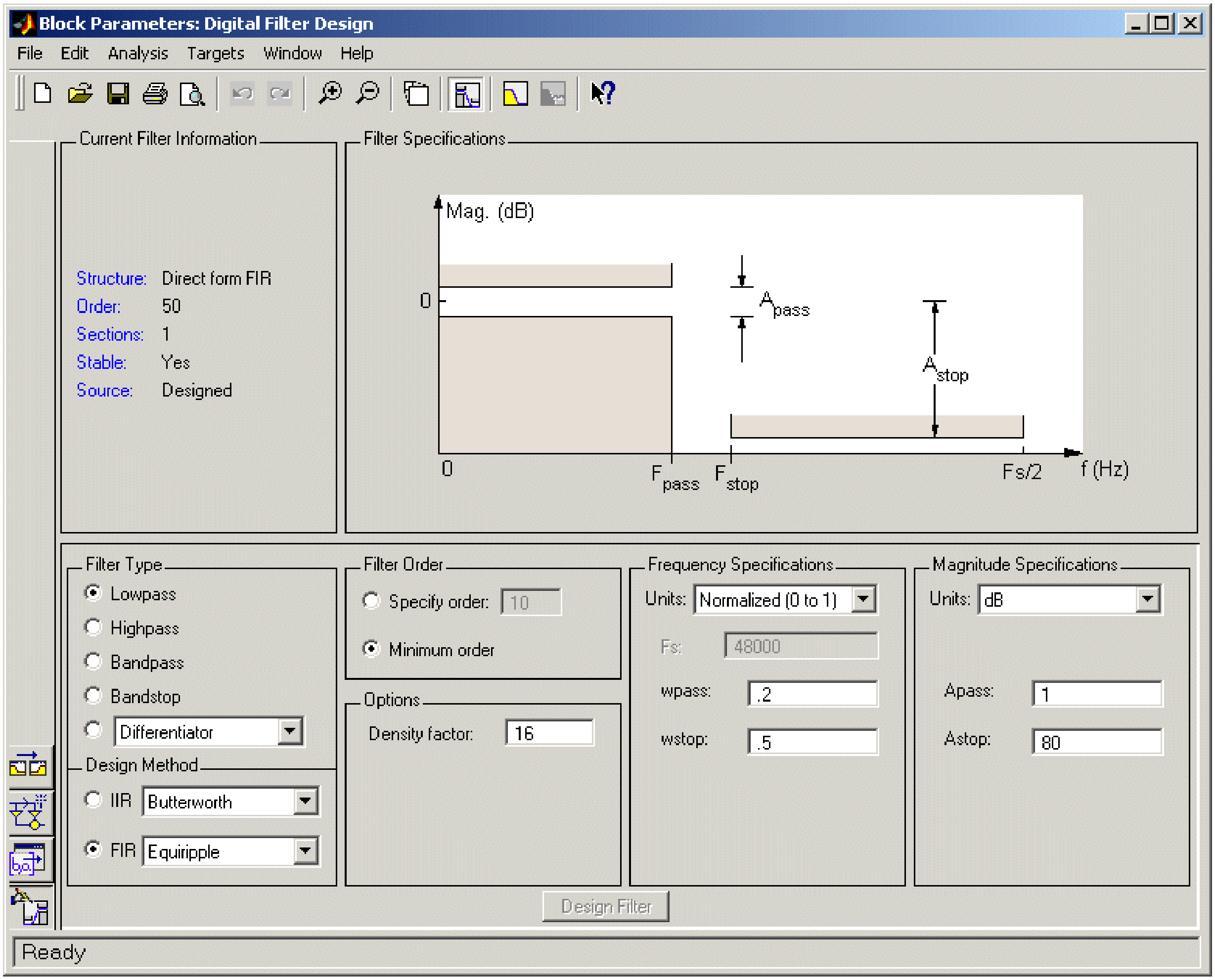


Model Using the Digital Filter Block to Implement Filters



Vector Scope Display After Running the Model

Dialog Box



Current Filter Information

Structure: Direct form FIR
Order: 50
Sections: 1
Stable: Yes
Source: Designed

Filter Specifications

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: .2
wstop: .5

Magnitude Specifications

Units: dB
Apass: 1
Astop: 80

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 the TI TMS320C6000™ DSP Platform.

To learn how to use the FDATool GUI, see “Designing the Filter” on page 7-213.

Supported Data Types

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

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

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:

- Chapter 3, “Filters” — Examples of when and how to use DSP Blockset filtering blocks
- “Digital Filter Design Block” on page 3-13
- Topic on the Filter Design and Analysis Tool (FDATool) in the Signal Processing Toolbox documentation.

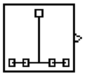
Purpose

Generate a discrete impulse

Library

DSP Sources

Description



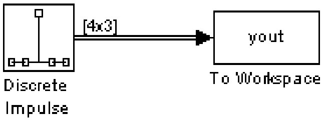
The Discrete Impulse block generates an impulse (the value `1`) at output sample `D+1`, where `D` is specified by the **Delay** parameter (`D` ≥ `0`). All output samples preceding and following sample `D+1` are zero.

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 $M*T_s$.

Examples

Construct the model below.



Configure the Discrete Impulse block to generate a frame-based three-channel output of type `double`, with impulses at samples 1, 4, and 6 of channels 1, 2, and 3, respectively. Use a sample period of 0.25 and a frame size of 4. The corresponding settings should be as follows:

- **Delay** = `[0 3 5]`
- **Sample time** = `0.25`
- **Samples per frame** = `4`
- **Output data type** = `double`

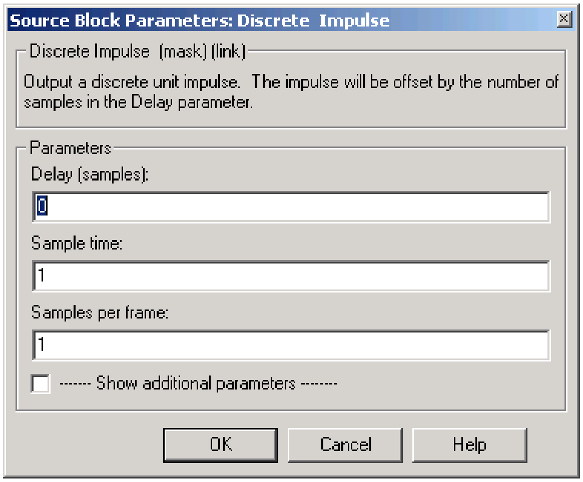
Run the model and look at the output, `yout`. The first few samples of each channel are shown below.

```
yout(1:10,:)
ans =

         1       0       0
```

```
0    0    0
0    0    0
0    1    0
0    0    0
0    0    1
0    0    0
0    0    0
0    0    0
0    0    0
```

The block generates an impulse at sample 1 of channel 1 (first column), at sample 4 of channel 2 (second column), and at sample 6 of channel 3 (third column).

Dialog Box



Source Block Parameters: Discrete Impulse

Discrete Impulse (mask) (link)

Output a discrete unit impulse. The impulse will be offset by the number of samples in the Delay parameter.

Parameters

Delay (samples): 0

Sample time: 1

Samples per frame: 1

□ ------- Show additional parameters --------

Delay

The number of zero-valued output samples, `D`, preceding the impulse. A length-N vector specifies an N-channel output. This parameter is not tunable.

Sample time

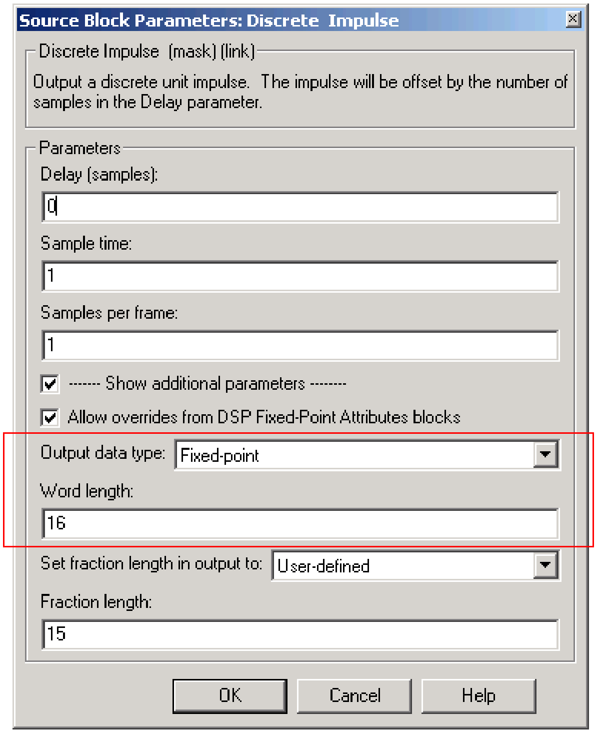
The sample period, T_s, of the output signal. The output frame period is $M*T_s$. This parameter is not tunable.

Samples per frame

The number of samples, M, in each output frame. This parameter is not tunable.

Show additional parameters

If selected, additional parameters specific to implementation of the block become visible as shown.



Source Block Parameters: Discrete Impulse

Discrete Impulse (mask) (link)

Output a discrete unit impulse. The impulse will be offset by the number of samples in the Delay parameter.

Parameters

Delay (samples): 0

Sample time: 1

Samples per frame: 1

☑ ------- Show additional parameters -------

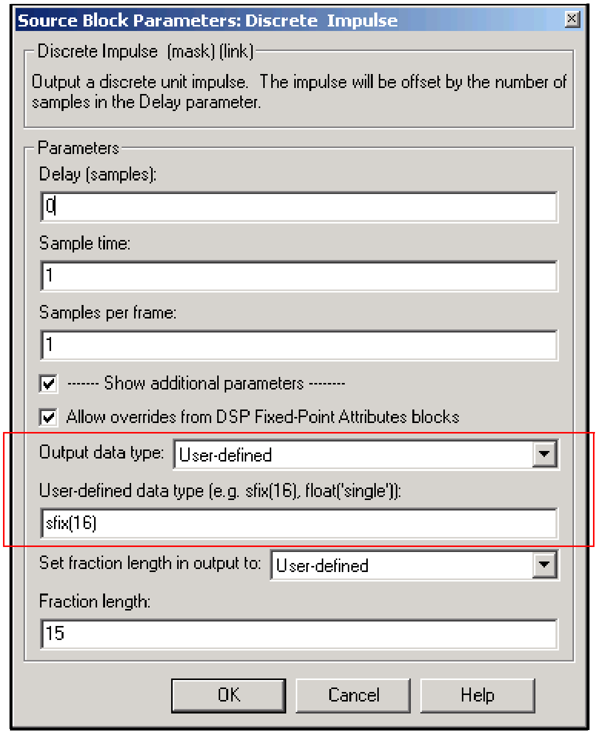
☑ Allow overrides from DSP Fixed-Point Attributes blocks

Output data type: Fixed-point

Word length: 16

Set fraction length in output to: User-defined

Fraction length: 15



Source Block Parameters: Discrete Impulse

Discrete Impulse (mask) (link)

Output a discrete unit impulse. The impulse will be offset by the number of samples in the Delay parameter.

Parameters

Delay (samples): 0

Sample time: 1

Samples per frame: 1

☑ ------- Show additional parameters -------

☑ Allow overrides from DSP Fixed-Point Attributes blocks

Output data type: User-defined

User-defined data type (e.g. sfix(16), float('single')): sfix(16)

Set fraction length in output to: User-defined

Fraction length: 15

Allow overrides from DSP Fixed-Point Attributes blocks

If you select this parameter, and if the **Output data type parameter** is set to `Fixed-point`, fixed-point data types for this block may be set by DSP Fixed-Point Attributes blocks in your model. If this parameter is unselected, the data types are always set by the parameters in the block mask.

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 fraction length in output to,** and **Fraction length** parameters.
- Choose `User-defined` to specify the output data type and scaling in the **User-defined data type**, **Set fraction length in output to,** and **Fraction length** parameters.
- Choose `Inherit via back propagation` to set the output data type and scaling to match the next block downstream.

Word length

Specify the word length, in bits, of the fixed-point output data type. This parameter is only visible 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 fraction length in output to

Specify the scaling of the fixed-point output by either of the following two methods:

- Choose `Best precision` to have the output scaling automatically set such that the output signal has the best possible precision.
- Choose `User-defined` to specify the output scaling in the **Fraction length** parameter.

This parameter is only visible if `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.

Fraction length

For fixed-point output data types, specify the number of fractional bits, or bits to the right of the binary point. This parameter is only visible if

`Fixed-point` or `User-defined` is selected for the **Output data type** parameter, and if `User-defined` is selected for the **Set fraction length in output to** parameter.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

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

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 "Creating Signals Using Signal Generator Blocks" on page 2-35 for how to use this and other blocks to generate signals.

Downsample

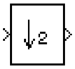
Purpose

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 $0 \leq D < (K-1)$, so that any of the K possible output phases can be selected. For example, when you downsample the sequence 1, 2, 3, ... by a factor of 4, you can select from the following four phases.

Input Sequence	Sample Offset, D	Output Sequence (K=4)
`1,2,3,...`	0	`0,1,5,9,13,17,21,25,...`
`1,2,3,...`	1	`0,2,6,10,14,18,22,26,...`
`1,2,3,...`	2	`0,3,7,11,15,19,23,27,...`
`1,2,3,...`	3	`0,4,8,12,16,20,24,28,...`

The initial zero in each output sequence above is a result of the default zero **Initial condition** parameter setting for this example. See “Latency” on page 7-225 for more on the **Initial condition** parameter.

This block supports triggered subsystems if, for **Sample-based mode**, you select `Force single-rate` and, for **Frame-based mode**, you select `Maintain input frame rate`.

Sample-Based Operation

When the input is sample based, the block treats each of the $M*N$ matrix elements as an independent channel, and downsamples each channel over time. The input and output sizes are identical.

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.

- `Force single rate`

  When `Force 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** pop-up menu does not affect sample-based inputs.

Frame-Based Inputs

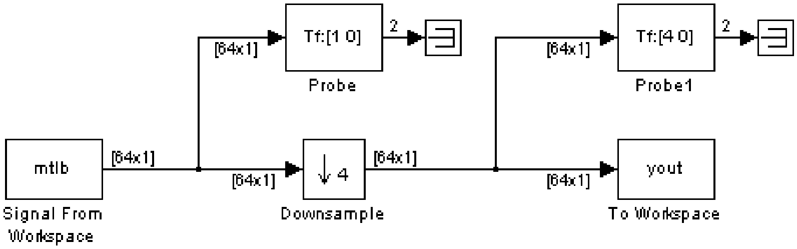
When the input is frame based, the block treats each of the N input columns as a frame containing M_i sequential time samples from an independent channel. The block downsamples each channel independently by discarding K-1 rows of the input matrix following each row that it passes through to the output.

The **Frame-based mode** parameter determines how the block adjusts the rate at the output to accommodate the reduced number of samples. There are two available options:

- `Maintain input frame size`

  The block generates the output at the slower (downsampled) rate by using a proportionally longer frame *period* at the output port than at the input port. For downsampling by a factor of K, the output frame period is K times longer than the input frame period (T_{fo} = KT_{fi}), but the input and output frame sizes are equal.

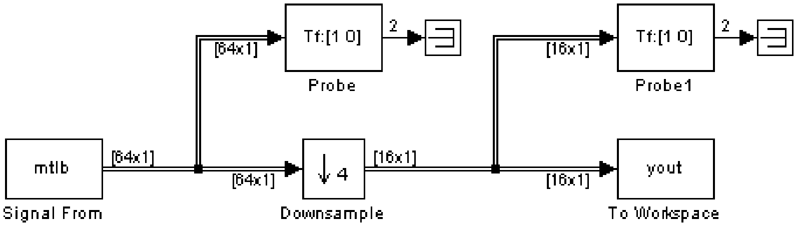
  The model below shows a single-channel input with a frame period of 1 second being downsampled by a factor of 4 to a frame period of 4 seconds. The input and output frame sizes are identical.



- `Maintain input frame rate`

  The block generates the output at the slower (downsampled) rate by using a proportionally smaller frame *size* than the input. For downsampling by a factor of K, the output frame size is K times smaller than the input frame size ($M_o = M_i/K$), but the input and output frame rates are equal.

  The model below shows a single-channel input of frame size 64 being downsampled by a factor of 4 to a frame size of 16. The input and output frame rates are identical.



The setting of the **Sample-based mode** pop-up menu does not affect frame-based inputs.

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 `Force single rate` is selected with D > 0. The latency in this case is one sample.

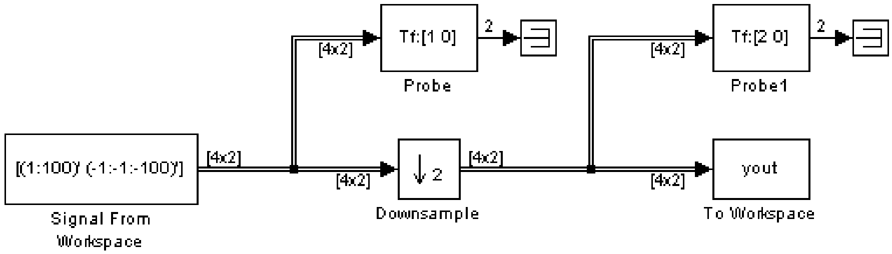
In all cases of *one-sample latency*, the initial condition for each channel appears as the first output sample. Input sample D+1 appears as the second output sample for each channel, followed by input sample D+1+K, input sample D+1+2K, and so on. The **Initial condition** parameter can be an M_i-by-N 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 2-98 for more information about block rates and the Simulink tasking modes.

Examples

Construct the frame-based model shown below.



Adjust the block parameters as follows:

- Configure the Signal From Workspace block to generate a two-channel signal with frame size of 4 and sample period of 0.25 second. This represents an output frame period of 1 second (0.25∗4). 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).

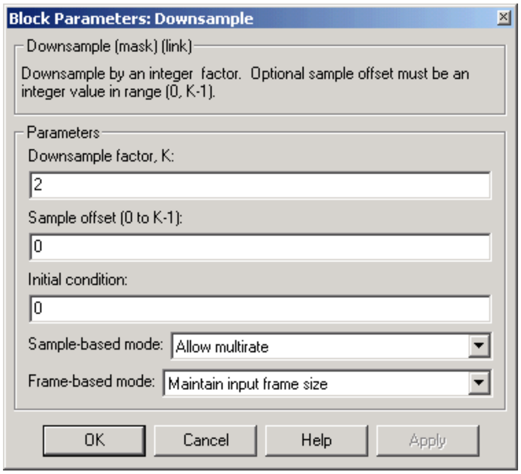
This model is multirate because there are at least two distinct frame rates, as shown by the two Probe blocks. To run this model in the Simulink multitasking mode, select `Fixed-step` and `discrete` from the **Type** controls in the **Solver** panel of the **Simulation Parameters** dialog box, and select `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 (that is, row D+1, where D is the **Sample offset**) appears in the output as sample 5 (that is sample M_i+1, where M_i is the input frame size).

Dialog Box



Block Parameters: Downsample

Downsample (mask) (link)

Downsample by an integer factor. Optional sample offset must be an integer value in range [0, K-1].

Parameters

Downsample factor, K: 2

Sample offset (0 to K-1): 0

Initial condition: 0

Sample-based mode: Allow multirate

Frame-based mode: Maintain input frame size

Downsample factor

The integer factor, K, by which to decrease the input sample rate.

Sample offset

The sample offset, D, which must be an integer in the range [0, K-1].

Initial condition

The value with which the block is initialized for cases of nonzero latency; a scalar or matrix.

Sample-based mode

The method by which to implement downsampling for sample-based inputs: `Allow multirate` (that is, decrease the output sample rate), or `Force single-rate` (that is, force the output sample rate to match the input sample rate by repeating every 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` (that is, decrease the frame rate), or `Maintain input frame rate` (that is, 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 your data types to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-2.

See Also

FIR Decimation	DSP Blockset
FIR Rate Conversion	DSP Blockset
Repeat	DSP Blockset
Sample and Hold	DSP Blockset
Upsample	DSP Blockset

DSP Constant

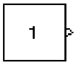
Purpose

Generate a discrete-time or continuous-time constant signal

Library

DSP Sources

Description

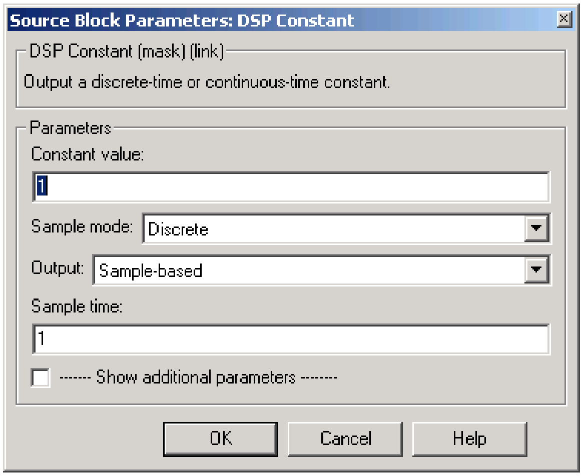


The DSP Constant block generates a signal whose value remains constant throughout the simulation. The **Constant value** parameter specifies the constant to output, and can be any valid MATLAB expression that evaluates to a scalar, vector, or matrix.

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



Source Block Parameters: DSP Constant

DSP Constant (mask) (link)

Output a discrete-time or continuous-time constant.

Parameters

Constant value: 1

Sample mode: Discrete

Output: Sample-based

Sample time: 1

□ ------- Show additional parameters --------

Opening this dialog box causes a running simulation to pause. See “Changing Source Block Parameters” in the online Simulink documentation for details.

Constant value

Specify the constant to generate. This parameter is tunable; values entered here can be tuned, but their dimensions must remain fixed.

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

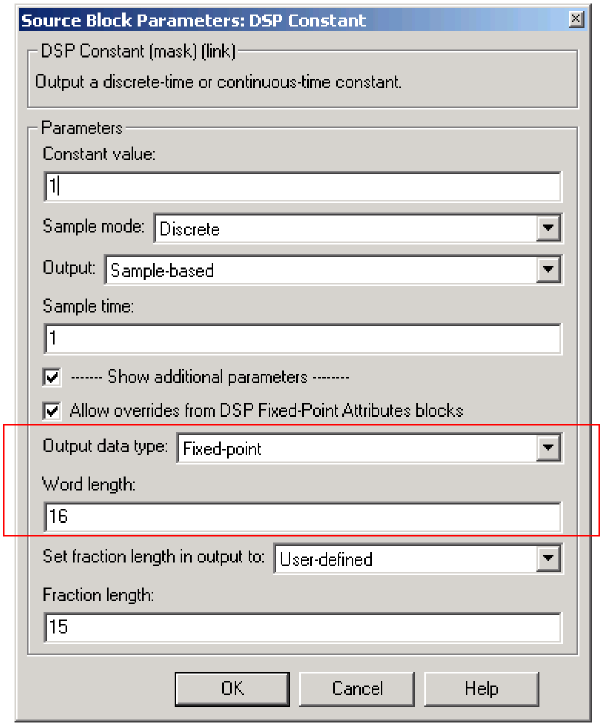
Specify whether the output is `Sample-based (interpret vectors as 1-D)`, `Sample-based`, or `Frame-based`. When `Sample-based` is selected and the output is a vector, its dimension is constrained to match the **Constant value** dimension (row or column). If `Sample-based (interpret vectors as 1-D)` is 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.



Source Block Parameters: DSP Constant

DSP Constant (mask) (link)

Output a discrete-time or continuous-time constant.

Parameters

Constant value: 1

Sample mode: Discrete

Output: Sample-based

Sample time: 1

☑ ------- Show additional parameters -------

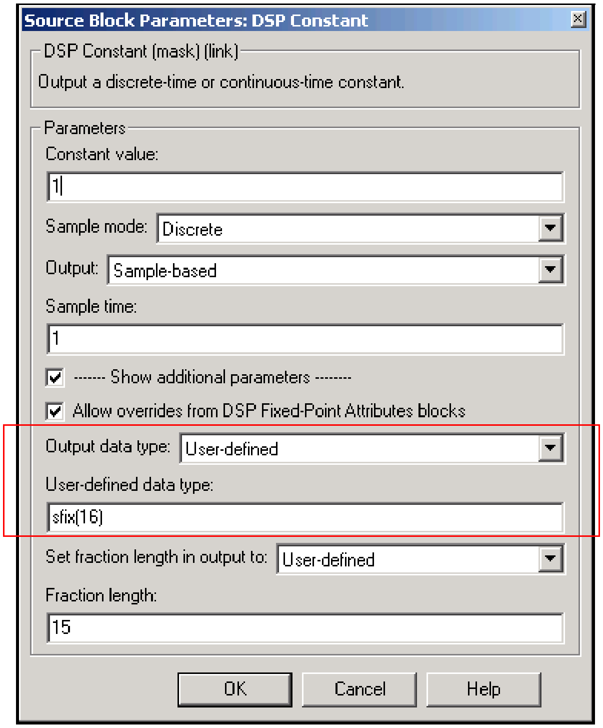
☑ Allow overrides from DSP Fixed-Point Attributes blocks

Output data type: Fixed-point

Word length: 16

Set fraction length in output to: User-defined

Fraction length: 15



Source Block Parameters: DSP Constant

DSP Constant (mask) (link)

Output a discrete-time or continuous-time constant.

Parameters

Constant value: 1

Sample mode: Discrete

Output: Sample-based

Sample time: 1

☑ ------- Show additional parameters -------

☑ Allow overrides from DSP Fixed-Point Attributes blocks

Output data type: User-defined

User-defined data type: sfix(16)

Set fraction length in output to: User-defined

Fraction length: 15

Allow overrides from DSP Fixed-Point Attributes blocks

If you select this parameter, and if the **Output data type parameter** is set to `Fixed-point`, fixed-point data types for this block may be set by DSP Fixed-Point Attributes blocks in your model. If this parameter is unselected, the data types are always set by the parameters in the block mask.

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 fraction length in output to,** and **Fraction length** parameters.

- Choose `User-defined` to specify the output data type and scaling in the **User-defined data type**, **Set fraction length in output to,** and **Fraction length** parameters.
- Choose `Inherit from `Constant value'` to set the output data type and scaling to match the values of the **Constant value** parameter.
- Choose `Inherit via back propagation` to set the output data type and scaling to match the following block.

The value of this parameter overrides any data type information specified in the **Constant value** parameter.

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 fraction length in output to

Specify the scaling of the fixed-point output by either of the following two methods:

- Choose `Best precision` to have the output scaling automatically set such that the output signal has the best possible precision.
- Choose `User-defined` to specify the output scaling in the **Fraction length** parameter.

This parameter is only visible if `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.

Fraction length

For fixed-point output data types, specify the number of fractional bits, or bits to the right of the binary point. This parameter is only visible if `Fixed-point` or `User-defined` is selected for the **Output data type** parameter, and if `User-defined` is selected for the **Set fraction length in output to** parameter.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

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

See Also

Constant	Simulink
Signal From Workspace	DSP Blockset

Also see “Creating Signals Using Constant Blocks” on page 2-32 for how to use this and other blocks to generate constant signals.

Purpose

Set fixed-point attributes of DSP Blockset blocks on the system or subsystem level

Library

- Signal Management / Signal Attributes
- Any library containing blocks that can be overridden by a DSP Fixed-Point Attributes block

Description

The DSP Fixed-Point Attributes (DFPA) block enables you to set fixed-point attributes for DSP Blockset blocks in your model on the system or subsystem level. This allows you to set fixed-point parameters for groups of blocks in one place, rather than on a block-by-block basis. The parameters listed below appear on various fixed-point DSP Blockset block masks, and can be controlled by DFPA blocks.

On nonsource blocks:

- **Output word length**
- **Output fraction length**
- **Accumulator word length**
- **Accumulator fraction length**
- **Product output word length**
- **Product output fraction length**
- **State memory word length**
- **State memory fraction length**
- **Round integer calculations toward**
- **Saturate on integer overflow**

On source blocks:

- **Word length**
- **Set fraction length in output to**
- **Fraction length**

The blocks that have parameters that may be controlled by DFPA blocks are listed below:

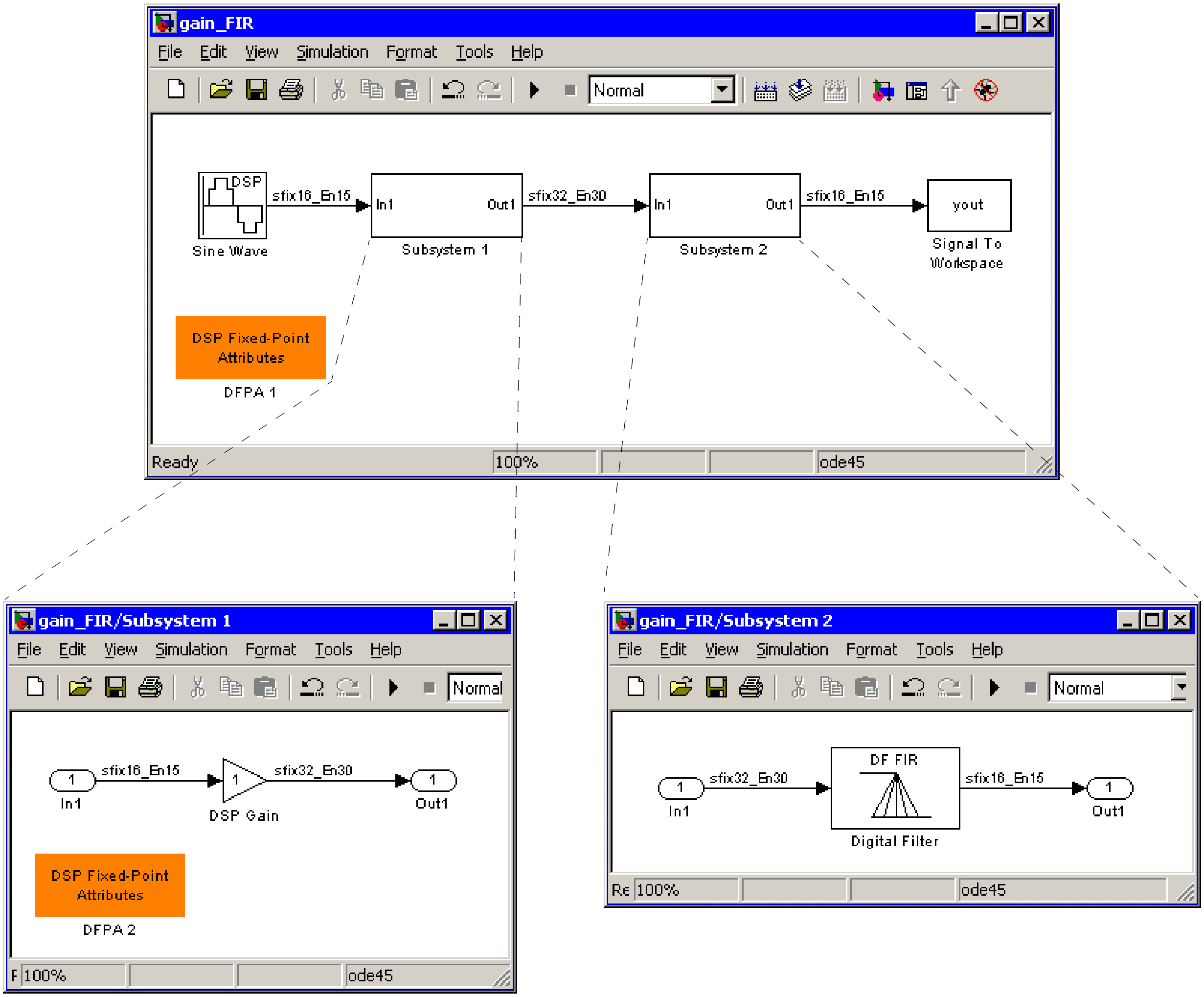
- Autocorrelation
- Constant Diagonal Matrix
- Convolution
- Correlation
- Digital Filter
- Discrete Impulse
- DSP Constant
- DSP Gain
- DSP Product
- DSP Sum
- FFT
- FIR Decimation
- FIR Interpolation
- Identity Matrix
- IFFT
- Matrix Product
- Matrix Scaling
- Matrix Sum
- Sine Wave
- Window Function

Each of these blocks has an **Allow overrides from DSP Fixed-Point Attributes blocks** check box that is selected by default. That means that any such blocks in models you build can automatically be configured from the top level of your model, without having to configure each block mask. If you do not want the fixed-point parameters of a particular block in your model to be controlled by DFPA blocks, clear the **Allow overrides from DSP Fixed-Point Attributes blocks** check box in that block's mask.

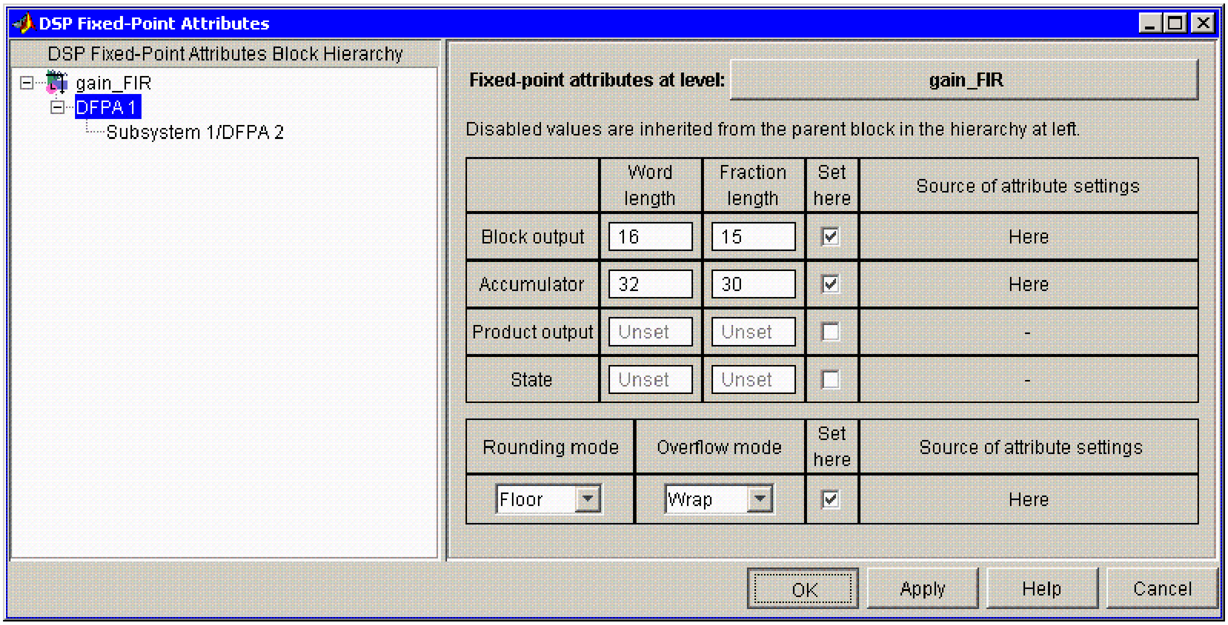
Place a DFPA block in any subsystem in your model that contains blocks with any of the fixed-point parameters listed above, if you want to control the blocks at the subsystem level. You can have a DFPA block in one, some, or all of the subsystems in your model. DFPA blocks in lower subsystems overrule the settings of DFPA blocks at higher levels.

Examples

Consider the following model `gain_FIR`, which contains `Subsystem 1` and `Subsystem 2`.



At the top level of the model, the DFPA 1 block is setting the block output and accumulator attributes for the system. The DFPA GUI for this block appears below.


DSP Fixed-Point Attributes
DSP Fixed-Point Attributes Block Hierarchy
OK
Apply
Help
Cancel

- gain_FIR
  - DFPA 1
    - Subsystem 1/DFPA 2

Fixed-point attributes at level: gain_FIR

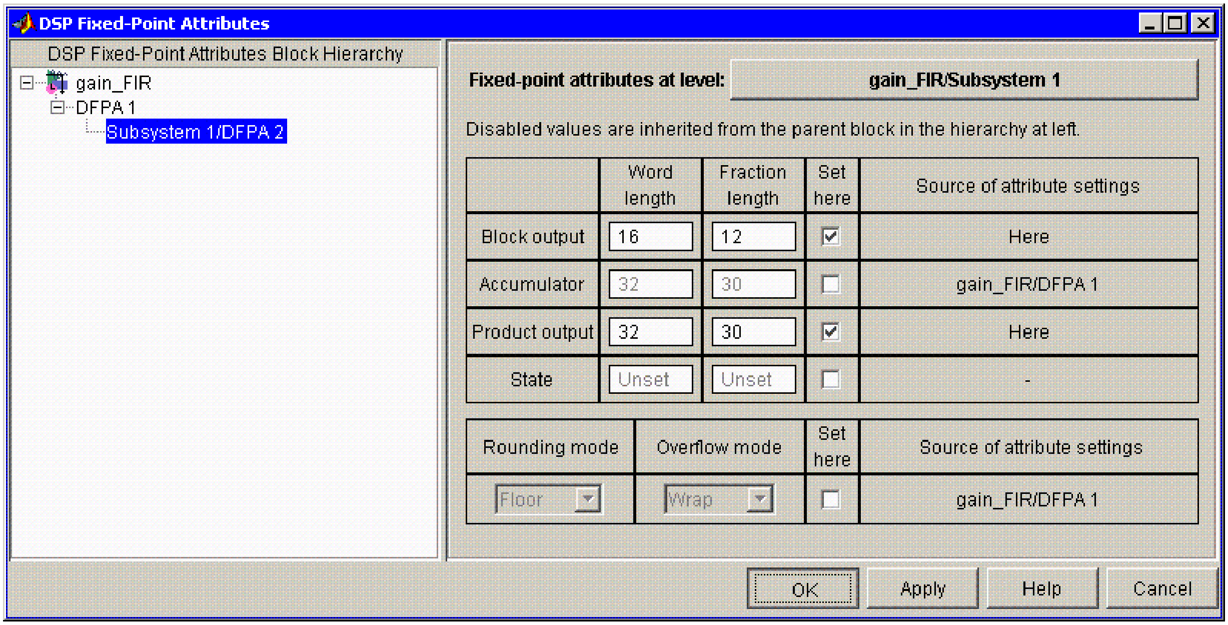
Disabled values are inherited from the parent block in the hierarchy at left.

	Word length	Fraction length	Set here	Source of attribute settings
Block output	16	15	☑	Here
Accumulator	32	30	☑	Here
Product output	Unset	Unset	□	-
State	Unset	Unset	□	-

Rounding mode	Overflow mode	Set here	Source of attribute settings
Floor	Wrap	☑	Here

Note that `DFPA 1` is highlighted in the DSP Fixed-Point Attributes Block Hierarchy in the left pane of the GUI. In the right pane, the attributes that the DFPA 1 block is setting for the model are shown. The **Set here** check box is selected in both the **Block output** and **Accumulator** rows, and the **Word length** and **Fraction length** parameters are set for those attributes. The **Set here** check box is also selected for the rounding and overflow modes, which are set to `Floor` and `Wrap`.

Subsystem 1 also contains a DFPA block. The settings for DFPA 2 are shown below.



DSP Fixed-Point Attributes Block Hierarchy

- gain_FIR
  - DFPA 1
    - Subsystem 1/DFPA 2

Fixed-point attributes at level: **gain_FIR/Subsystem 1**

Disabled values are inherited from the parent block in the hierarchy at left.

	Word length	Fraction length	Set here	Source of attribute settings
Block output	16	12	☑	Here
Accumulator	32	30	□	gain_FIR/DFPA 1
Product output	32	30	☑	Here
State	Unset	Unset	□	-

Rounding mode	Overflow mode	Set here	Source of attribute settings
Floor	Wrap	□	gain_FIR/DFPA 1

Note that `Subsystem 1/DFPA 2` is highlighted in the DSP Fixed-Point Attributes Block Hierarchy in the left pane of the GUI. In the right pane, the attributes that the DFPA 2 block is setting for the model are shown. The **Set here** check box is selected in both the **Block output** and **Product output** rows, and the **Word length** and **Fraction length** parameters are set for those attributes. Here, the DFPA 2 block is overriding the block output parameters set by the DFPA 1 block for Subsystem 1. The blocks in Subsystem 2, however, are still being controlled only by DFPA 1.

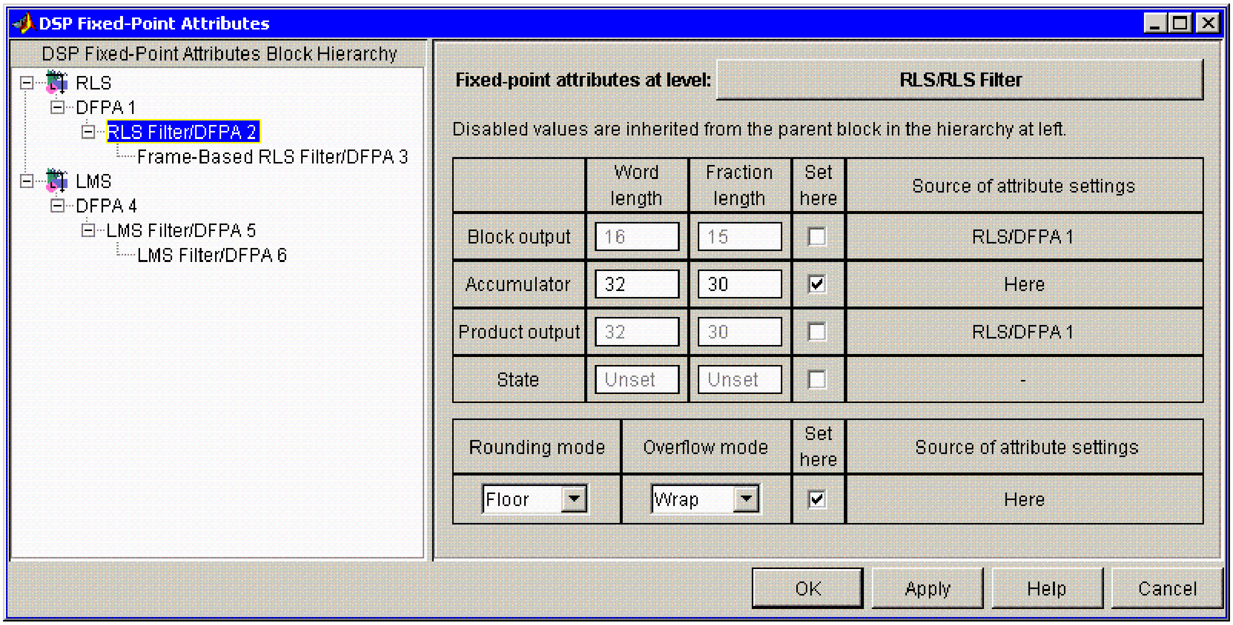
The following table lists each of the blocks in the model that is controllable by DFPA blocks, and shows from where each applicable block parameter is being set.

Block	Attribute	Parameters Set In
Sine Wave		
	block output	DFPA 1
	accumulator	DFPA 1
	product output	unset/set in block mask
	state	unset/set in block mask
	rounding and overflow modes	DFPA 1
DSP Gain		
	block output	DFPA 2
	accumulator	DFPA 1
	product output	DFPA 2
	state	unset/set in block mask
	rounding and overflow modes	DFPA 1
Digital Filter		
	block output	DFPA 1
	accumulator	DFPA 1
	product output	unset/set in block mask
	state	unset/set in block mask
	rounding and overflow modes	DFPA 1

Note The block output of a DSP Gain block is controlled by the **Product output** attributes on a DFPA block. The **Block output** attributes on a DFPA block are ignored for DSP Gain blocks.

Dialog Box

When you double-click on a DFPA block in any model you have open, the DSP Fixed-Point Attributes GUI appears. This one instance of the GUI enables you to see the information for all DFPA blocks in open models. You can see the hierarchy of DFPA blocks in the left pane of the GUI, and settings for a particular DFPA block in the right pane of the GUI.



DSP Fixed-Point Attributes

DSP Fixed-Point Attributes Block Hierarchy

- RLS
  - DFPA 1
    - RLS Filter/DFPA 2
      - Frame-Based RLS Filter/DFPA 3
- LMS
  - DFPA 4
    - LMS Filter/DFPA 5
      - LMS Filter/DFPA 6

Fixed-point attributes at level: RLS/RLS Filter

Disabled values are inherited from the parent block in the hierarchy at left.

	Word length	Fraction length	Set here	Source of attribute settings
Block output	16	15	□	RLS/DFPA 1
Accumulator	32	30	☑	Here
Product output	32	30	□	RLS/DFPA 1
State	Unset	Unset	□	-

Rounding mode	Overflow mode	Set here	Source of attribute settings
Floor	Wrap	☑	Here

Left Pane

The left pane of the DSP Fixed-Point Attributes GUI displays the DSP Fixed-Point Attributes Block Hierarchy. This navigation tree displays the relative hierarchy of all DFPA blocks in models that are currently open.

Note The left pane displays the hierarchy of *DFPA blocks* in all models that are currently open. It does not display the hierarchy of *subsystems* in your models.

The top-level nodes in the hierarchy, designated by the icon, represent each model that you have open. The branches under each top-level model node show the DFPA blocks in that model. Settings in DFPA blocks that are at a lower level in the hierarchy have precedence over higher-level DFPA blocks for the subsystems that they control. Therefore, a DFPA block controls fixed-point settings for blocks that are in the same subsystem or a lower subsystem, unless

- A lower-level DFPA block overrides the settings.
- A particular fixed-point block does not have its **Allow overrides from DSP Attributes blocks** check box selected.

You can click on any branch in the hierarchy to select it. The information for the DFPA block selected in the left pane is displayed in the right pane.

Right Pane

The following buttons, rows, and columns allow you to specify the settings for the currently selected DFPA block:

Fixed-point attributes at level:

This button displays the path to the subsystem that contains the DFPA block currently selected in the DFPA Block Hierarchy. Click this button to bring the subsystem to the front of your screen.

Block output

This row allows you to set fixed-point block output attributes.

This row can override the **Output word length** and **Output fraction length** parameters on nonsource block masks. However, note that the settings in this row are ignored by the DSP Gain, DSP Product, and DSP Sum blocks.

This row can override the **Word length** and **Fraction length** parameters on source masks, if the **Output data type** parameter of the source block is set to `Fixed-point`.

The block output word length and fraction length may only be set in this row if the corresponding **Set here** check box is selected.

Accumulator

This row allows you to set fixed-point accumulator attributes. This row can override the **Accumulator word length** and **Accumulator fraction length** parameters on block masks. This row also overrides the **Output word length** and **Output fraction length** parameters for DSP Sum blocks.

The accumulator word length and fraction length may only be set in this row if the corresponding **Set here** check box is selected.

Product Output

This row allows you to set fixed-point product output attributes. This row can override the **Product output word length** and **Product output fraction length** parameters on block masks. This row also overrides the **Output word length** and **Output fraction length** parameters for DSP Gain and DSP Product blocks.

The product output word length and fraction length may only be set in this row if the corresponding **Set here** check box is selected.

State

This row allows you to set fixed-point state memory attributes. This row can override the **State memory word length** and **State memory output fraction length** parameters on block masks.

The state word length and fraction length may only be set in this row if the corresponding **Set here** check box is selected.

Rounding Mode

This column allows you to set the fixed-point rounding mode to `Floor` or `Nearest`. This column can override the **Round integer calculations toward** parameter on block masks.

The rounding mode may only be set in this column if the corresponding **Set here** check box is selected.

Overflow Mode

This column allows you to set the fixed-point overflow mode to `Wrap` or `Saturate`. This column can override the **Saturate on integer overflow** parameter on block masks.

The overflow mode may only be set in this column if the corresponding **Set here** check box is selected.

Word length

This column allows you to set the word length, in bits, for the Block output, Accumulator, Product output, and State attributes. Each word length must be an integer number of bits between 2 and 128.

Fraction length

This column allows you to set the fraction length, in bits, for the Block output, Accumulator, Product output, and State attributes. Each fraction length must be an integer number of bits.

Set here

The check boxes in this column allow you to specify whether each attribute is set by the currently selected DFPA block. If a **Set here** check box is not selected, the corresponding attribute is either set by a higher DFPA block, or is not set at all.

Source of attribute settings

This column indicates the level at which each attribute is set. If the **Set here** check box is selected for a certain row, the **Source of attribute settings** cell in that row is set to `Here`. If the **Set here** check box is not selected for a certain row, the **Source of attribute settings** cell for that row gives the path to the DFPA block controlling those attributes for the current subsystem. If the parameters in a row are not set in any DFPA block, the **Source of attribute settings** cell for that row has a dash `-`.

Supported Data Types

This block sets attributes for DSP Blockset blocks that perform fixed-point calculations. This block has no input or output ports.

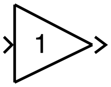
Purpose

Multiply the input by a constant

Library

Math Functions / Math Operations

Description



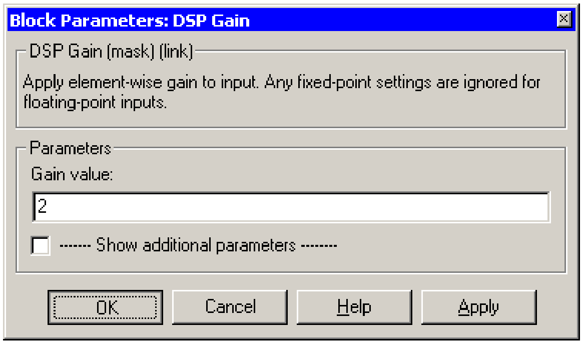
The DSP Gain block is a masked version of the Simulink Gain block. It multiplies the input by the gain, element-wise.

The input and the gain can each be a scalar, vector, or matrix. Either the gain must be a scalar, or it must have the same dimensions as the input:

- If the gain is a scalar, then the gain is multiplied to each element of the input.
- If the gain is a vector or a matrix, each element of the gain is multiplied to the corresponding element of the input.

The DSP Gain block accepts real and complex floating-point and fixed-point inputs.

Dialog Box


Block Parameters: DSP Gain

DSP Gain (mask) (link)

Apply element-wise gain to input. Any fixed-point settings are ignored for floating-point inputs.

Parameters

Gain value:

2

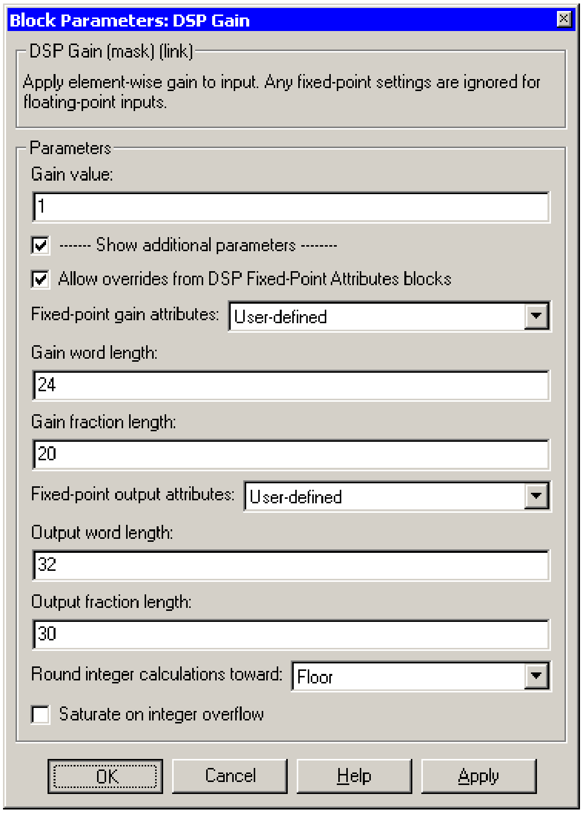
□ ------- Show additional parameters --------

Gain value

Specify the value by which to multiply the input. The gain may be scalar, vector, or a matrix. The gain may not be Boolean.

Show additional parameters

If selected, additional parameters specific to implementation of the block become visible as shown.



DSP Gain (mask) (link)

Apply element-wise gain to input. Any fixed-point settings are ignored for floating-point inputs.

Parameters

Gain value: 1

☑ ------- Show additional parameters -------

☑ Allow overrides from DSP Fixed-Point Attributes blocks

Fixed-point gain attributes: User-defined

Gain word length: 24

Gain fraction length: 20

Fixed-point output attributes: User-defined

Output word length: 32

Output fraction length: 30

Round integer calculations toward: Floor

□ Saturate on integer overflow

Allow overrides from DSP Fixed-Point Attributes blocks

If you select this parameter, fixed-point data types for this block may be set by DSP Fixed-Point Attributes (DFPA) blocks in your model. If this parameter is unselected, the data types are always set by the parameters in the block mask.

Note that the data type of the gain is always specified by the value in the DSP Gain block mask, and not by a DFPA block.

When a DSP Gain block is overridden by a DFPA block, the output data type of the DSP Gain block is controlled by the **Product output** attribute. The **Output** attribute of the DFPA block is ignored for DSP Gain blocks.

Fixed-point gain attributes

Choose how you will specify the word length and fraction length of the gain. If you select `Same as input`, these characteristics will match those of the input to the block. If you select `User-defined`, the **Gain word length** and **Gain fraction length** parameters become visible.

The Gain does not obey the **Round integer calculations toward** and **Saturate on integer overflow** parameters; it is always saturated and rounded to `Nearest`.

Gain word length

Specify the word length, in bits, of the gain. This parameter is only visible if `User-defined` is specified for the **Fixed-point gain attributes** parameter.

Gain fraction length

Specify the fraction length, in bits, of the gain. This parameter is only visible if `User-defined` is specified for the **Fixed-point gain attributes** parameter.

Fixed-point output attributes

Choose how you will specify the output word length and fraction length. If you select `Same as input`, these characteristics will match those of the input to the block. If you select `User-defined`, the **Output word length** and **Output fraction length** parameters become visible.

Output word length

Specify the word length, in bits, of the output. This parameter is only visible if `User-defined` is specified for the **Fixed-point output attributes** parameter.

Output fraction length

Specify the fraction length, in bits, of the output. This parameter is only visible if `User-defined` is specified for the **Fixed-point output attributes** parameter.

Round integer calculations toward

Select the rounding mode for fixed-point operations. The gain does not obey this parameter; it always rounds to `Nearest`.

Saturate on integer overflow

If selected, overflows saturate. The gain does not obey this parameter; it is always saturated.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point
- 8-, 16-, and 32-bit signed integers

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

See Also

DSP Product	DSP Blockset
DSP Sum	DSP Blockset
Gain	Simulink

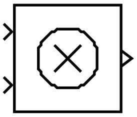
Purpose

Perform element-wise multiplication of two inputs

Library

Math Functions / Math Operations

Description



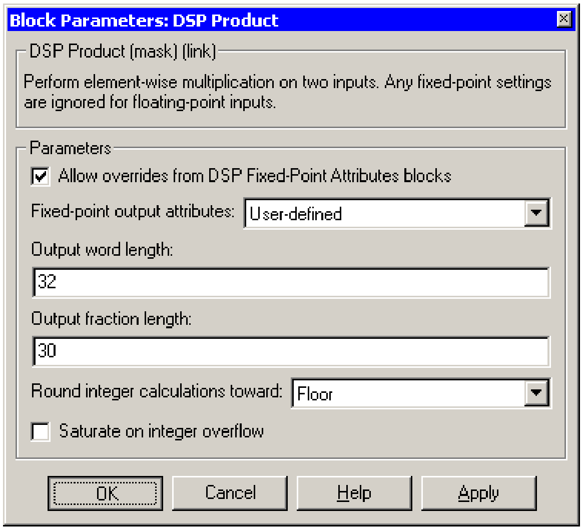
The DSP Product block is a masked version of the Simulink Product block. The DSP Product block performs element-wise multiplication of two inputs.

The inputs to the DSP Product block can be scalar, vector, or matrix. Either both inputs must have the same dimensions, or at least one of the inputs must be a scalar:

- If one input is a scalar, it is multiplied to each element of the other input.
- If both inputs are vectors or a matrices, their corresponding elements are multiplied together.

The DSP Product block accepts real and complex floating-point and fixed-point inputs.

Dialog Box



DSP Product (mask) (link)

Perform element-wise multiplication on two inputs. Any fixed-point settings are ignored for floating-point inputs.

Parameters

☑ Allow overrides from DSP Fixed-Point Attributes blocks

Fixed-point output attributes: User-defined

Output word length: 32

Output fraction length: 30

Round integer calculations toward: Floor

☐ Saturate on integer overflow

Allow overrides from DSP Fixed-Point Attributes blocks

If you select this parameter, fixed-point data types for this block may be set by DSP Fixed-Point Attributes (DFPA) blocks in your model. If this parameter is unselected, the data types are always set by the parameters in the block mask.

When a DSP Product block is overridden by a DFPA block, the output data type of the DSP Product block is controlled by the **Product output** attribute. The **Output** attribute of the DFPA block is ignored for DSP Product blocks.

Fixed-point output attributes

Choose how you will specify the output word length and fraction length. If you select `Same as first input`, these characteristics will match those of the first input to the block. If you select `User-defined`, the **Output word length** and **Output fraction length** parameters become visible.

Output word length

Specify the word length, in bits, of the output. This parameter is only visible if `User-defined` is specified for the **Fixed-point output attributes** parameter.

Output fraction length

Specify the fraction length, in bits, of the output. This parameter is only visible if `User-defined` is specified for the **Fixed-point output attributes** parameter.

Round integer calculations toward

Select the rounding mode for fixed-point operations.

Saturate on integer overflow

If selected, overflows saturate.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point
- 8-, 16-, and 32-bit signed integers

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

See Also

DSP Gain	DSP Blockset
DSP Sum	DSP Blockset
Product	Simulink

DSP Sum

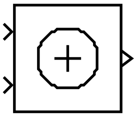
Purpose

Add two inputs

Library

Math Functions / Math Operations

Description



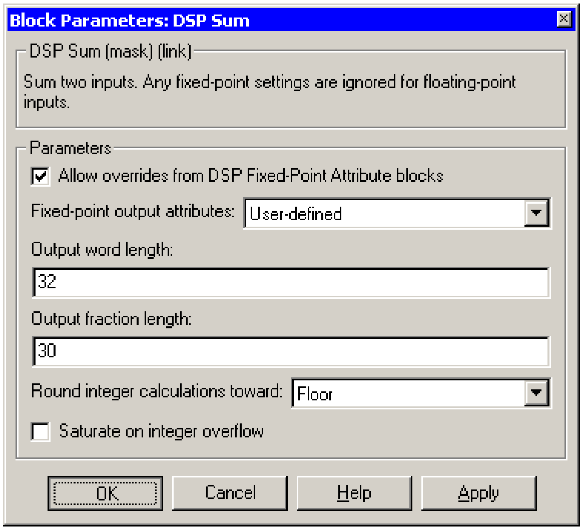
The DSP Sum block is a masked version of the Simulink Sum block. This block adds two inputs.

The inputs to the DSP Sum block can be scalar, vector, or matrix. Either both inputs must have the same dimensions, or at least one of the inputs must be a scalar:

- If one input is a scalar, it is added to each element of the other input.
- If both inputs are vectors or a matrices, their corresponding elements are added together.

The DSP Sum block accepts real and complex floating-point and fixed-point inputs.

Dialog Box



Block Parameters: DSP Sum

DSP Sum (mask) (link)

Sum two inputs. Any fixed-point settings are ignored for floating-point inputs.

Parameters

☑ Allow overrides from DSP Fixed-Point Attribute blocks

Fixed-point output attributes: User-defined

Output word length: 32

Output fraction length: 30

Round integer calculations toward: Floor

☐ Saturate on integer overflow

Allow overrides from DSP Fixed-Point Attributes blocks

If you select this parameter, fixed-point data types for this block may be set by DSP Fixed-Point Attributes (DFPA) blocks in your model. If this parameter is unselected, the data types are always set by the parameters in the block mask.

When a DSP Sum block is overridden by a DFPA block, the output data type of the DSP Sum block is controlled by the **Accumulator** attribute. The **Output** attribute of the DFPA block is ignored for DSP Sum blocks.

Fixed-point output attributes

Choose how you will specify the output word length and fraction length. If you select `Same as first input`, these characteristics will match those of the first input to the block. If you select `User-defined`, the **Output word length** and **Output fraction length** parameters become visible.

Output word length

Specify the word length, in bits, of the output. This parameter is only visible if `User-defined` is specified for the **Fixed-point output attributes** parameter.

Output fraction length

Specify the fraction length, in bits, of the output. This parameter is only visible if `User-defined` is specified for the **Fixed-point output attributes** parameter.

Round integer calculations toward

Select the rounding mode for fixed-point operations.

Saturate on integer overflow

If selected, overflows saturate.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point
- 8-, 16-, and 32-bit signed integers

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

See Also

DSP Gain	DSP Blockset
DSP Product	DSP Blockset
Sum	Simulink

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 block reference page for more information on how to use the block.

The DWT block computes the discrete wavelet transform (DWT) of each column of a frame-based input. By default, the output is a sample-based vector or matrix with the same dimensions as the input. Each column of the output is the DWT of the corresponding input column.

You must install the Wavelet Toolbox for the block to automatically design wavelet-based filters to compute the DWT. Otherwise, you must specify your own lowpass and highpass FIR filters by setting the **Filter** parameter to **User defined**.

For detailed information about how to use this block, see the Dyadic Analysis Filter Bank block reference page.

Examples

See “Examples” on page 7-266 in the Dyadic Analysis Filter Bank block reference page.

See Also

Dyadic Analysis Filter Bank	DSP Blockset

Dyadic Analysis Filter Bank

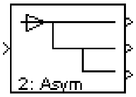
Purpose

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.)

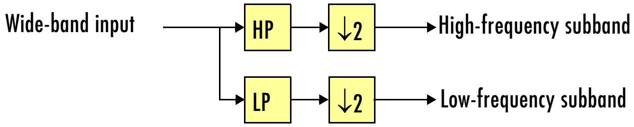
The Dyadic Analysis Filter Bank block decomposes a broadband signal into a collection of subbands with smaller bandwidths and slower sample rates. The block uses a series of highpass and lowpass FIR filters to repeatedly divide the input frequency range, as illustrated in the n-Level Asymmetric Dyadic Analysis Filter Bank figure.

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

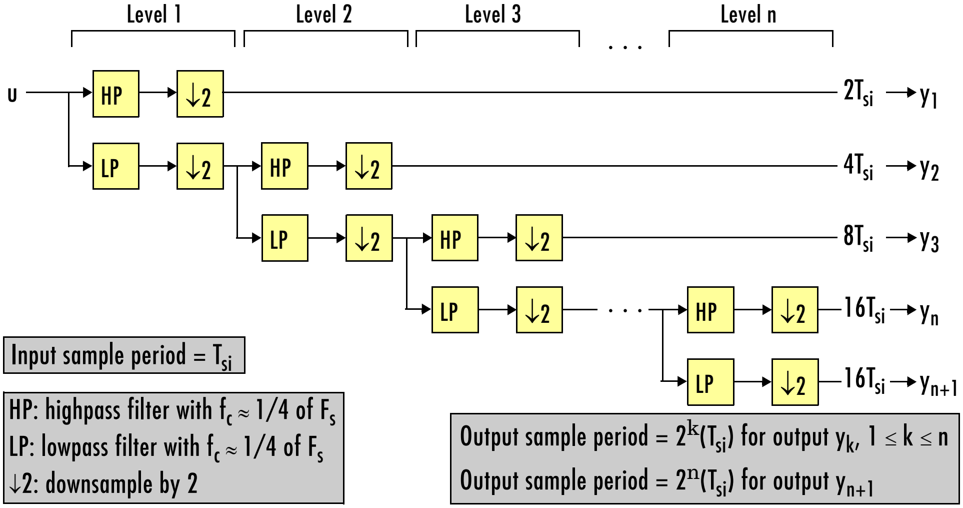
Dyadic analysis filter banks are constructed from the following basic unit. The unit can be cascaded to construct dyadic analysis filter banks with either a symmetric or asymmetric tree structure.



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.

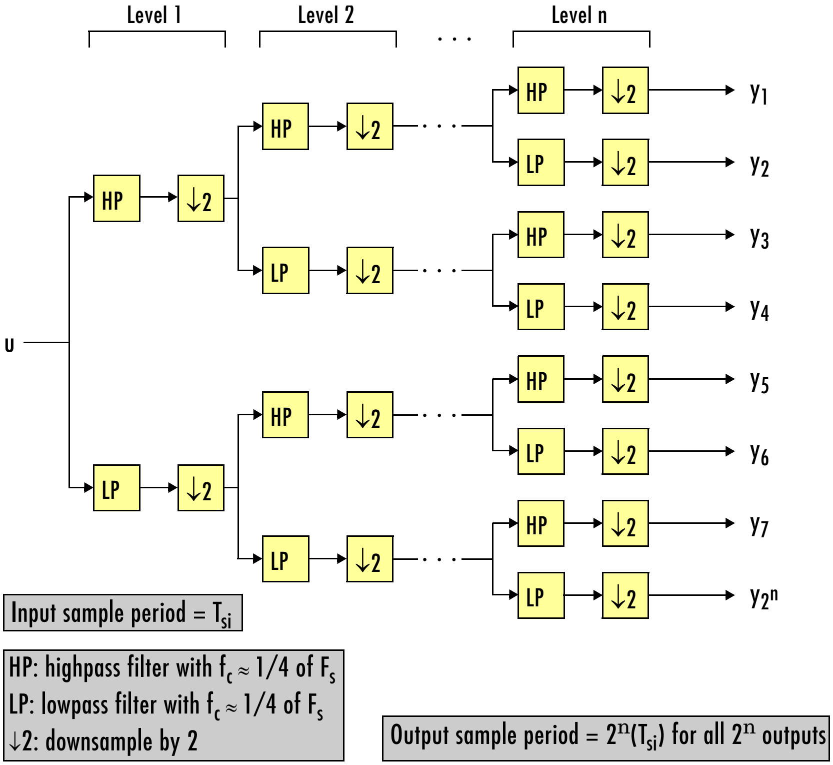
The unit decomposes its input into adjacent high-frequency and low-frequency subbands. Compared to the input, each subband has half the bandwidth (due to the half-band filters) and half the sample rate (due to the decimation by 2).

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*.


Level 1
Level 2
Level 3
. . .
Level n
u
HP
↓2
LP
$2T_{si}$
y_1
$4T_{si}$
y_2
$8T_{si}$
y_3
$16T_{si}$
y_n
$16T_{si}$
y_{n+1}
Input sample period = T_{si}
HP: highpass filter with $f_c \approx 1/4$ of F_s
LP: lowpass filter with $f_c \approx 1/4$ of F_s
↓2: downsample by 2
Output sample period = $2^k(T_{si})$ for output y_k, $1 \le k \le n$
Output sample period = $2^n(T_{si})$ for output y_{n+1}

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.


Level 1
Level 2
. . .
Level n
u
HP
LP
↓2
y_1
y_2
y_3
y_4
y_5
y_6
y_7
y_{2^n}
Input sample period = T_{si}
HP: highpass filter with $f_c \approx 1/4$ of F_s
LP: lowpass filter with $f_c \approx 1/4$ of F_s
↓2: downsample by 2
Output sample period = $2^n(T_{si})$ for all 2^n outputs

n-Level Symmetric Dyadic Analysis Filter Bank

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

Notable Characteristics of Asymmetric and Symmetric Dyadic Analysis Filter Banks

	n-level Symmetric	n-level Asymmetric
Low- and High-Frequency Subband Decomposition	All the low-frequency and high-frequency subbands in a level are decomposed in the next level.	Each level's low-frequency subband is decomposed in the next level, and each level's high-frequency band is an output of the filter bank.
Number of Output Subbands	2^n	n+1

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

	n-level Symmetric	n-level Asymmetric
Bandwidth and Number of Samples in Output Subbands	For an input with bandwidth BW and N samples, all outputs have bandwidth BW / 2^n and N / 2^n samples.	For an input with bandwidth BW and N samples, y_k has the bandwidth BW_k, and N_k samples, where $$BW_k = \begin{cases} BW/2^k & (1 \le k \le n) \\ BW/2^n & (k = n+1) \end{cases}$$ $$N_k = \begin{cases} N/2^k & (1 \le k \le n) \\ N/2^n & (k = n+1) \end{cases}$$ The bandwidth of, and number of samples in each subband (except the last) is half those of the previous subband. The last two subbands have the same bandwidth and number of samples since they originate from the same level in the filter bank.
Output Sample Period	All output subbands have a sample period of $2^n(T_{si})$	Sample period of kth output $$= \begin{cases} 2^k(T_{si}) & (1 \le k \le n) \\ 2^n(T_{si}) & (k = n+1) \end{cases}$$ Due to the decimations by 2, the sample period of each subband (except the last) is twice that of the previous subband. The last two subbands have the same sample period since they originate from the same level in the filter bank.

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

	n-level Symmetric	n-level Asymmetric
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.

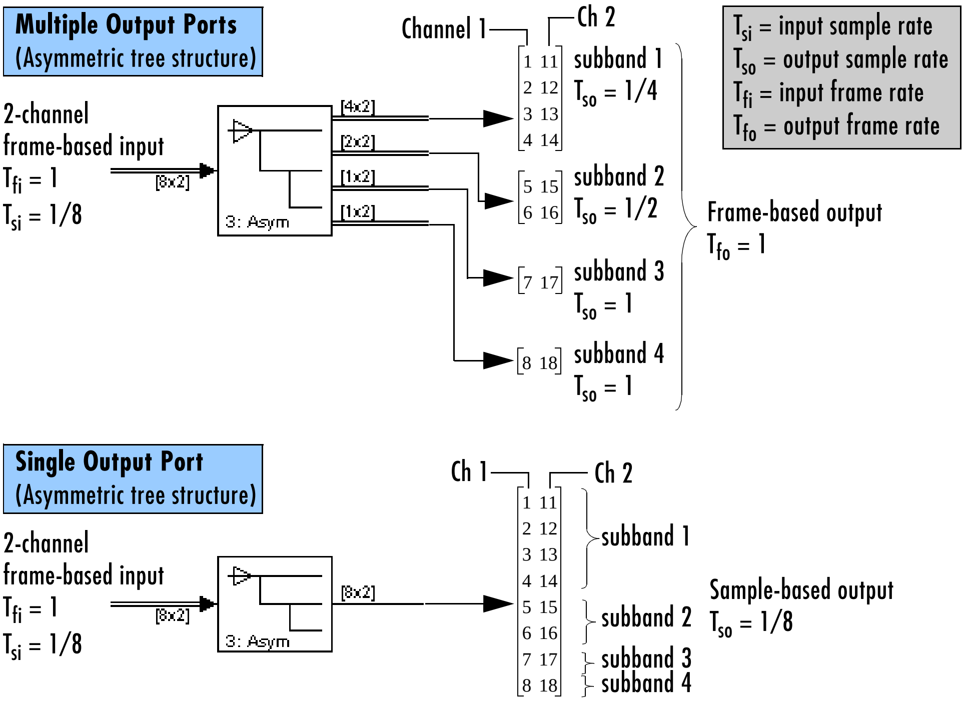
For an illustration of why the above input requirements exist, see the following figure called Outputs of a 3-Level Asymmetric Dyadic Analysis Filter Bank.

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 the following table called Output Characteristics for n-level Dyadic Analysis Filter Bank.

For more information about the filter bank levels and structures, see "Review of Dyadic Analysis Filter Banks" on page 7-257.



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.

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).

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 $\mathrm{M_i}$ output y_k has frame size $\mathrm{M}_{\mathrm{o},k}$: • `Symmetric` — All outputs have the frame size, $\mathrm{M_i} / 2^{\mathrm{n}}$ • `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, $\mathrm{F_{so,k}}$, are different from the input sample rate, $\mathrm{F_{si}}$: • `Symmetric` — All outputs have the sample rate $\mathrm{F_{si}} / 2^{\mathrm{n}}$. • `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.

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')`

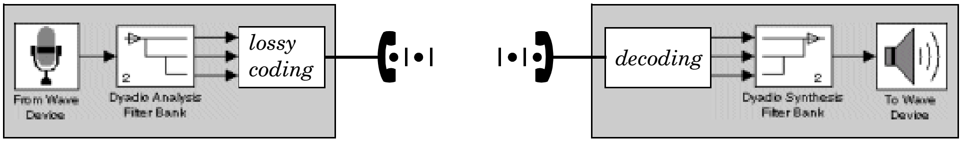
Specifying Filters with the Filter Parameter and Related Parameters (Continued)

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)

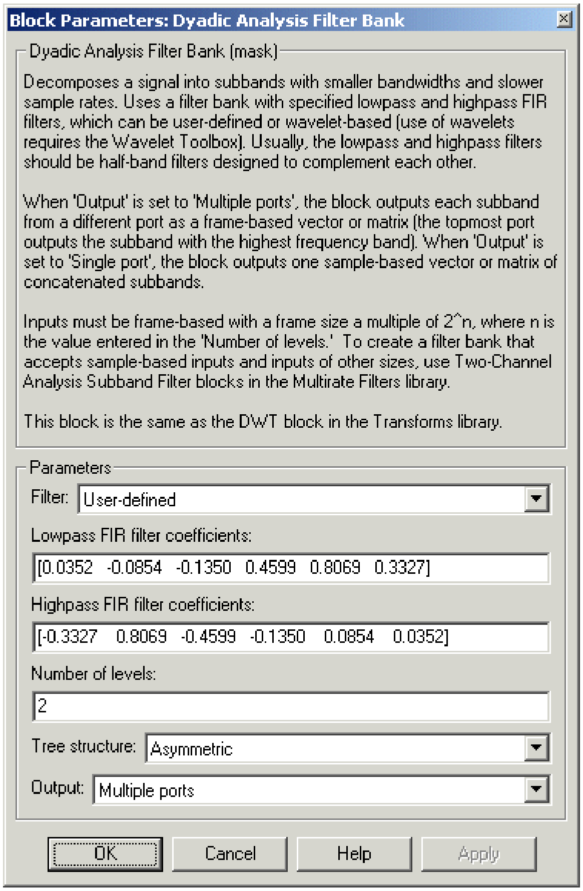
Note To see the version of the demos that use the Dyadic Analysis Filter Bank and Dyadic Synthesis Filter Bank blocks, click the **Frame-Based Demo** button in the demos.

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.



Block Parameters: Dyadic Analysis Filter Bank

Dyadic Analysis Filter Bank (mask)

Decomposes a signal into subbands with smaller bandwidths and slower sample rates. Uses a filter bank with specified lowpass and highpass FIR filters, which can be user-defined or wavelet-based (use of wavelets requires the Wavelet Toolbox). Usually, the lowpass and highpass filters should be half-band filters designed to complement each other.

When 'Output' is set to 'Multiple ports', the block outputs each subband from a different port as a frame-based vector or matrix (the topmost port outputs the subband with the highest frequency band). When 'Output' is set to 'Single port', the block outputs one sample-based vector or matrix of concatenated subbands.

Inputs must be frame-based with a frame size a multiple of 2^n, where n is the value entered in the 'Number of levels.' To create a filter bank that accepts sample-based inputs and inputs of other sizes, use Two-Channel Analysis Subband Filter blocks in the Multirate Filters library.

This block is the same as the DWT block in the Transforms library.

Parameters

Filter: User-defined

Lowpass FIR filter coefficients:

[0.0352 -0.0854 -0.1350 0.4599 0.8069 0.3327]

Highpass FIR filter coefficients:

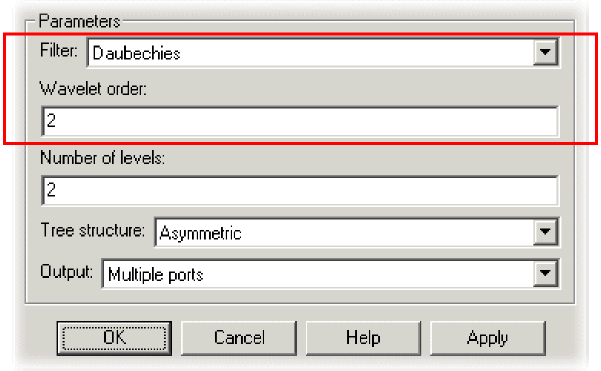
[-0.3327 0.8069 -0.4599 -0.1350 0.0854 0.0352]

Number of levels:

2

Tree structure: Asymmetric

Output: Multiple ports



Parameters

Filter: Daubechies

Wavelet order:

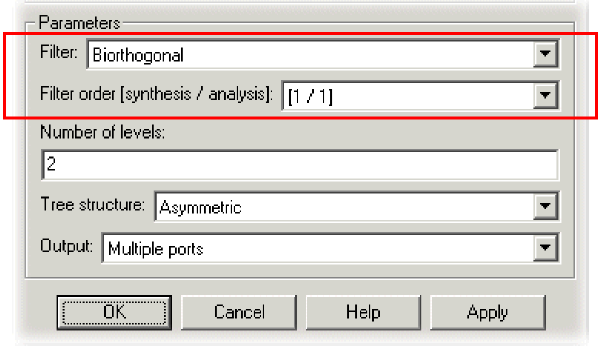
2

Number of levels:

2

Tree structure: Asymmetric

Output: Multiple ports



Parameters

Filter: Biorthogonal

Filter order [synthesis / analysis]: [1 / 1]

Number of levels:

2

Tree structure: Asymmetric

Output: Multiple ports

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 the Specifying Filters with the Filter Parameter and Related Parameters table.

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 the Specifying Filters with the Filter Parameter and Related Parameters table.

Filter order [synthesis / analysis]

The order of the wavelet for the synthesis and analysis filter stages. For example, 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 the Specifying Filters with the Filter Parameter and Related Parameters table.

Number of levels

The number of filter bank levels. An n-level asymmetric structure has n+1 outputs, and an n-level symmetric structure has 2^n outputs, as shown in the figures n-Level Asymmetric Dyadic Analysis Filter Bank and n-Level Symmetric Dyadic Analysis Filter Bank. The block's icon displays the value of this parameter in the lower left corner.

Tree structure

The structure of the filter bank: `Asymmetric`, or `Symmetric`. See the figures entitled n-Level Asymmetric Dyadic Analysis Filter Bank and n-Level Symmetric Dyadic Analysis Filter Bank.

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 7-261.

References

Fliege, N. J. *Multirate Digital Signal Processing: Multirate Systems, Filter Banks, Wavelets*. West Sussex, England: John Wiley & Sons, 1994.

Strang, G. and T. Nguyen. *Wavelets and Filter Banks*. Wellesley, MA: Wellesley-Cambridge Press, 1996.

Vaidyanathan, P. P. *Multirate Systems and Filter Banks*. Englewood Cliffs, NJ: Prentice Hall, 1993.

Supported Data Types

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

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

See Also

Dyadic Synthesis Filter Bank	DSP Blockset
Two-Channel Analysis Subband Filter	DSP Blockset

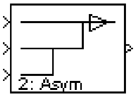
Purpose

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 the figure entitled n-Level Asymmetric Dyadic Synthesis Filter Bank. 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.

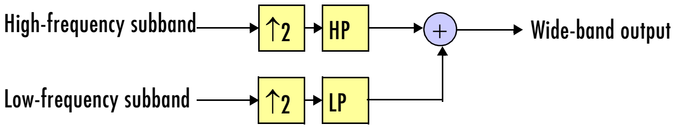
This block automatically computes wavelet-based perfect reconstruction filters if the wavelet selection in the **Filter** parameter of this block is the *same* as the **Filter** parameter setting of the corresponding Dyadic Analysis Filter Bank block. The use of wavelets requires the Wavelet Toolbox. To learn how to design your own perfect reconstruction filters, see "References" on page 7-285.

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

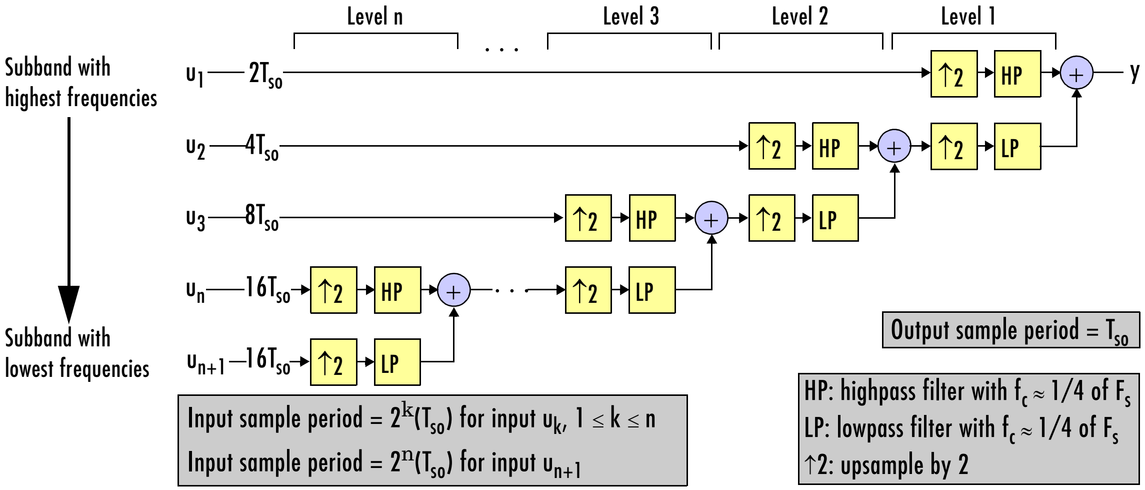
Dyadic synthesis filter banks are constructed from the following basic unit. The unit can be cascaded to construct dyadic synthesis filter banks with either a asymmetric or symmetric tree structure as illustrated in the figures entitled n-Level Asymmetric Dyadic Synthesis Filter Bank and n-Level Symmetric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



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

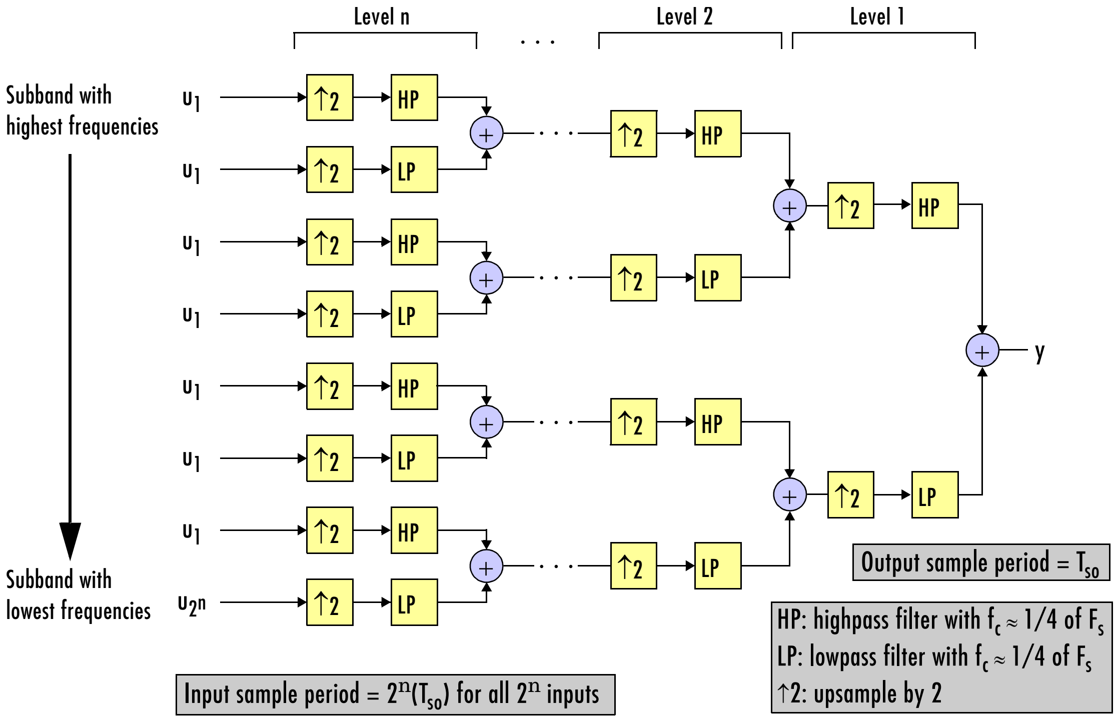
The unit takes in adjacent high-frequency and low-frequency subbands, and reconstructs them into a wide-band signal. Compared to each subband input, the output has twice the bandwidth and twice the sample rate.

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


Level n
. . .
Level 3
Level 2
Level 1
Subband with highest frequencies
Subband with lowest frequencies
u_1 — $2T_{so}$
u_2 — $4T_{so}$
u_3 — $8T_{so}$
u_n — $16T_{so}$
u_{n+1} — $16T_{so}$
↑2
HP
LP
+
y
Output sample period = T_{so}
Input sample period = $2^k(T_{so})$ for input u_k, $1 \le k \le n$
Input sample period = $2^n(T_{so})$ for input u_{n+1}
HP: highpass filter with $f_c \approx 1/4$ of F_s
LP: lowpass filter with $f_c \approx 1/4$ of F_s
↑2: upsample by 2

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.


Level n
. . .
Level 2
Level 1
Subband with highest frequencies
Subband with lowest frequencies
u1
u1
u1
u1
u1
u1
u1
u2n
↑2
HP
LP
+
y
Output sample period = Tso
HP: highpass filter with fc ≈ 1/4 of Fs
LP: lowpass filter with fc ≈ 1/4 of Fs
↑2: upsample by 2
Input sample period = 2n(Tso) for all 2n inputs

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.

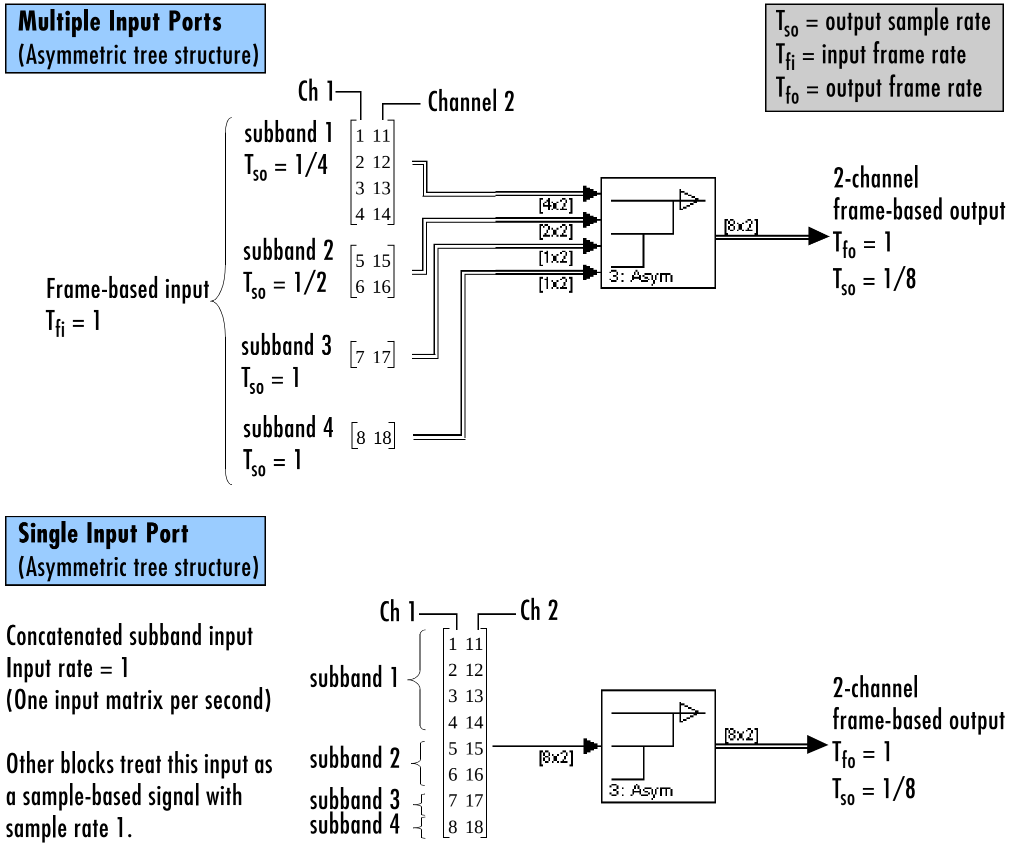
Note Any output of a Dyadic Analysis Filter Bank block whose parameter settings match the corresponding settings of this block is a valid input to this block. For example, the setting of the Dyadic Analysis Filter Bank block parameter, **Output**, must be the same as this block's **Input** parameter (`Single port` or `Multiple ports`).

Valid Inputs for Input Set to Single Port

- Inputs must be sample-based vectors or sample-based matrices of concatenated subbands.
- Each input column contains the subbands for an independent signal.
- Upper input rows contain the high-frequency subbands, and the lower rows contain the low-frequency subbands.

Valid Inputs for Input Set to Multiple Ports

- Inputs must be a frame-based vector or frame-based matrix for each subband, each of which is input to a separate input port.
- The columns of each input contains a subband for an independent signal.
- The input to the topmost input port is the subband containing the highest frequencies, and the input to the bottommost port is the subband containing the lowest frequencies.



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 7-272.

Output Characteristics

The following table summarizes the output characteristics for both frame-based inputs, and concatenated subband inputs. For an illustration of why the output characteristics exist, see the figure called Valid Inputs to a 3-Level Asymmetric Dyadic Synthesis Filter Bank.

	Frame-Based Inputs (Input = Multiple ports)	**Concatenated Subband Inputs (Input = Single port)**
Output Frame Status	Outputs are always frame based regardless of the input frame status. Each output column is an independent channel, reconstructed from the corresponding channel in the inputs.	
Output Frame Rate	Same as the input frame rate.	Same as the input rate (the rate of the concatenated subband inputs).
Output Frame Dimensions	• The output has the same number of columns as the inputs. • The number of output rows depends on the tree structure of the filter bank: - **Asymmetric** — The number of output rows is twice the number of rows in the input to the topmost input port. - **Symmetric** — The number of output rows is the product of the number of input ports and the number of rows in an input to any input port.	The output has the same number of rows and columns as the input.

For general information about the filter banks, see “Review of Dyadic Synthesis Filter Banks” on page 7-272.

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 7-285.
- 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.

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')`

Specifying Filters with the Filter Parameter and Related Parameters (Continued)

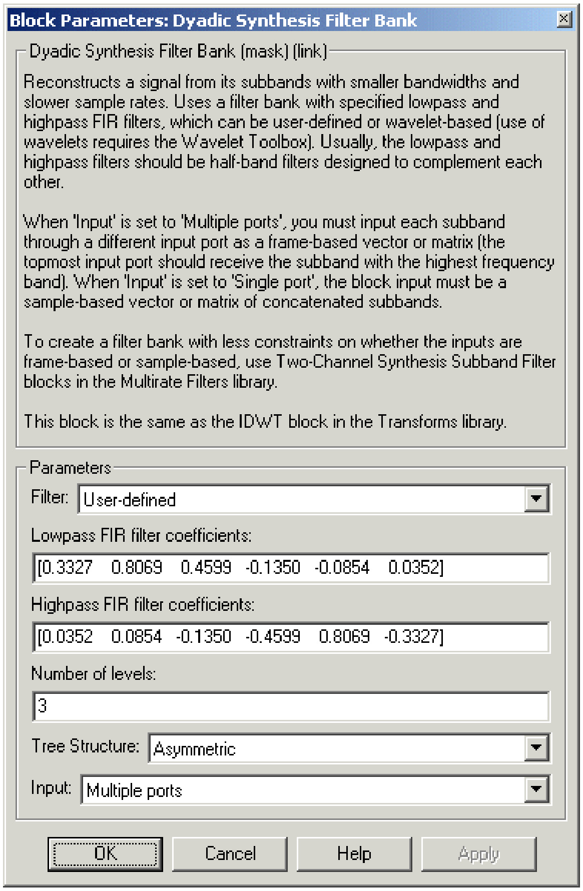
Filter	Sample Setting for Related Filter Specification Parameters	Corresponding Wavelet Function Syntax
Symlets	**Wavelet order =** `3`	`wfilters('sym3')`
Coiflets	**Wavelet order =** `1`	`wfilters('coif1')`
Biorthogonal	**Filter order [synthesis / analysis] =** `[3/1]`	`wfilters('bior3.1')`
Reverse Biorthogonal	**Filter order [synthesis / analysis] =** `[3/1]`	`wfilters('rbio3.1')`
Discrete Meyer	None	`wfilters('dmey')`

Examples

See “Examples” on page 7-266 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.



Block Parameters: Dyadic Synthesis Filter Bank

Dyadic Synthesis Filter Bank (mask) (link)

Reconstructs a signal from its subbands with smaller bandwidths and slower sample rates. Uses a filter bank with specified lowpass and highpass FIR filters, which can be user-defined or wavelet-based (use of wavelets requires the Wavelet Toolbox). Usually, the lowpass and highpass filters should be half-band filters designed to complement each other.

When 'Input' is set to 'Multiple ports', you must input each subband through a different input port as a frame-based vector or matrix (the topmost input port should receive the subband with the highest frequency band). When 'Input' is set to 'Single port', the block input must be a sample-based vector or matrix of concatenated subbands.

To create a filter bank with less constraints on whether the inputs are frame-based or sample-based, use Two-Channel Synthesis Subband Filter blocks in the Multirate Filters library.

This block is the same as the IDWT block in the Transforms library.

Parameters

Filter: User-defined

Lowpass FIR filter coefficients:

[0.3327 0.8069 0.4599 -0.1350 -0.0854 0.0352]

Highpass FIR filter coefficients:

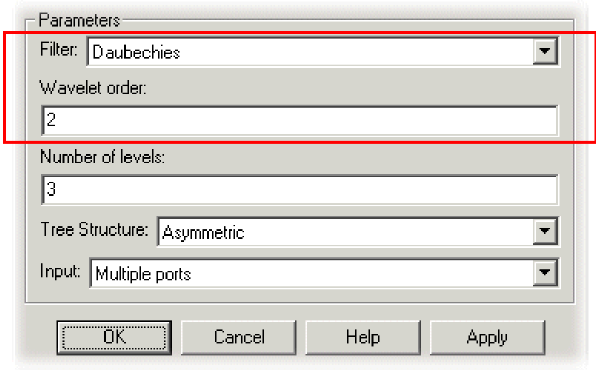
[0.0352 0.0854 -0.1350 -0.4599 0.8069 -0.3327]

Number of levels:

3

Tree Structure: Asymmetric

Input: Multiple ports



Parameters

Filter: Daubechies

Wavelet order:

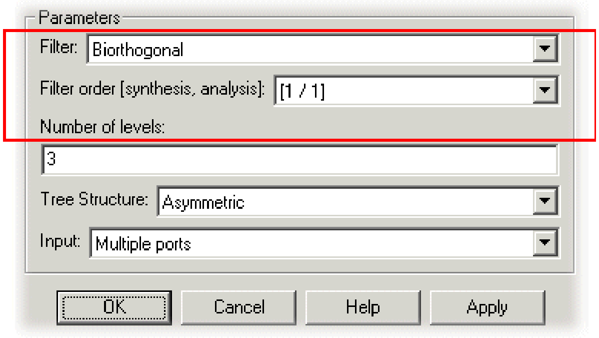
2

Number of levels:

3

Tree Structure: Asymmetric

Input: Multiple ports



Parameters

Filter: Biorthogonal

Filter order [synthesis, analysis]: [1 / 1]

Number of levels:

3

Tree Structure: Asymmetric

Input: Multiple ports

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 the table called Specifying Filters with the Filter Parameter and Related Parameters.

Lowpass FIR filter coefficients

A vector of filter coefficients (descending powers of z) that specifies coefficients used by all the lowpass filters in the filter bank. This parameter is enabled when you set **Filter** to `User defined`. The lowpass filter should be a half-band filter that passes the frequency band stopped by the filter specified in the **Highpass FIR filter coefficients** parameter. To perfectly reconstruct a signal decomposed by the Dyadic Analysis Filter Bank, the filters in this block *must* be designed to perfectly reconstruct the outputs of the analysis filter bank. Otherwise, the reconstruction will not be perfect. The default values of this parameter specify a perfect reconstruction filter for the default settings of the Dyadic Analysis Filter Bank (based on a Daubechies wavelet with wavelet order `3`).

Highpass FIR filter coefficients

A vector of filter coefficients (descending powers of z) that specifies coefficients used by all the highpass filters in the filter bank. This parameter is enabled when you set **Filter** to `User defined`. The highpass filter should be a half-band filter that passes the frequency band stopped by the filter specified in the **Lowpass FIR filter coefficients** parameter. To perfectly reconstruct a signal decomposed by the Dyadic Analysis Filter Bank, the filters in this block *must* be designed to perfectly reconstruct the outputs of the analysis filter bank. Otherwise, the reconstruction will not be perfect. The default values of this parameter specify a perfect reconstruction filter for the default settings of the Dyadic Analysis Filter Bank (based on a Daubechies wavelet with wavelet order `3`).

Wavelet order

The order of the wavelet selected in the **Filter** parameter. This parameter is enabled only when you set **Filter** to certain types of wavelets, as shown in the table called Specifying Filters with the Filter Parameter and Related Parameters.

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 in the table called Specifying Filters with the Filter Parameter and Related Parameters.

Number of levels

The number of filter bank levels. An n-level asymmetric structure has n+1 outputs, and an n-level symmetric structure has 2^n outputs, as shown in the figures entitled n-Level Asymmetric Dyadic Synthesis Filter Bank and n-Level Symmetric Dyadic Synthesis Filter Bank. The block's icon displays the value of this parameter in the lower-left corner.

Tree structure

The structure of the filter bank: `Asymmetric`, or `Symmetric`. See the figures entitled n-Level Asymmetric Dyadic Synthesis Filter Bank and n-Level Symmetric Dyadic Synthesis Filter Bank.

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 7-277.

References

Fliege, N. J. *Multirate Digital Signal Processing: Multirate Systems, Filter Banks, Wavelets*. West Sussex, England: John Wiley & Sons, 1994.

Strang, G. and T. Nguyen. *Wavelets and Filter Banks*. Wellesley, MA: Wellesley-Cambridge Press, 1996.

Vaidyanathan, P. P. *Multirate Systems and Filter Banks*. Englewood Cliffs, NJ: Prentice Hall, 1993.

Supported Data Types

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

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

See Also

Dyadic Analysis Filter Bank	DSP Blockset
Two-Channel Synthesis Subband Filter	DSP Blockset

See “Multirate Filters” on page 3-61 for related information.

Edge Detector

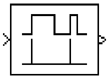
Purpose

Detect a transition of the input from zero to a nonzero value

Library

Signal Management / Switches and Counters

Description

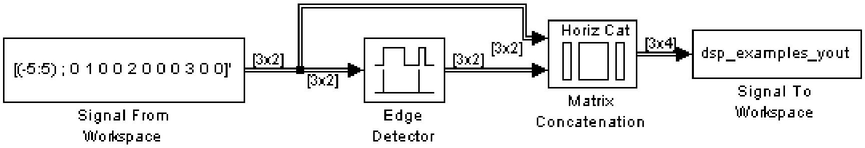


The Edge Detector block generates an impulse (the value `1`) in a given output channel when the corresponding channel of the input transitions from zero to a nonzero value. Otherwise, the block generates zeros in each channel.

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 (that is, a zero at the bottom of the first frame, and a nonzero value at the top of the following frame) is counted in the frame that contains the nonzero value.

Examples

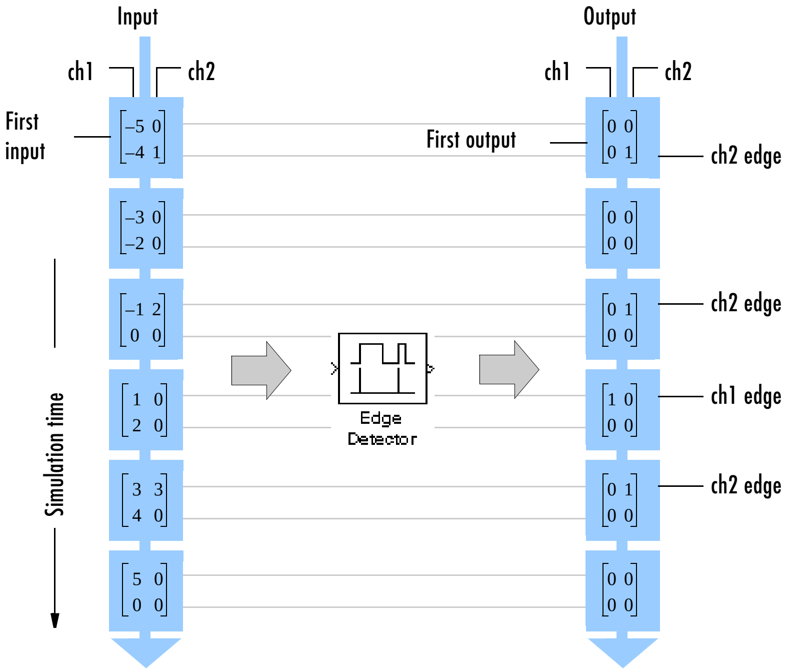
In the model below, the Edge Detector block locates the edges (zero to nonzero transitions) in a two-channel frame-based input with frame size 3. The two input channels are horizontally concatenated with the two output channels to create the four-channel workspace variable `dsp_examples_yout`.



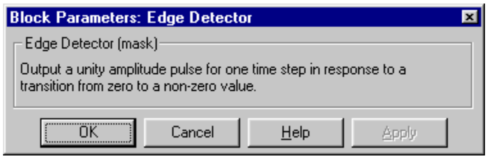
Adjust the block parameters as described below. (Use the default settings for the To Workspace block.)

- Set the Signal From Workspace block parameters as follows:
  - **Signal** = `[(-5:5) ; 0 1 0 0 2 0 0 0 3 0 0]'`
  - **Sample time** = `1`
  - **Samples per frame** = `3`
- Set the Matrix Concatenation block parameters as follows:
  - **Number of inputs** = `2`
  - **Concatenation method** = `Horizontal`

As shown below, the block finds edges at sample 7 in channel 1, and at samples 2, 5, and 9 in channel 2.



Dialog Box


Block Parameters: Edge Detector

Edge Detector (mask)

Output a unity amplitude pulse for one time step in response to a transition from zero to a non-zero value.

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-7. To learn how to disable Boolean output support, see “Steps to Disabling Boolean Support” on page A-8.
- 8-, 16-, and 32-bit signed integers

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

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

See Also

Counter	DSP Blockset
Event-Count Comparator	DSP Blockset

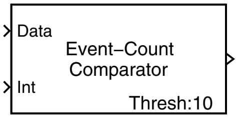
Purpose

Detect threshold crossing of accumulated nonzero inputs

Library

Signal Management / Switches and Counters

Description



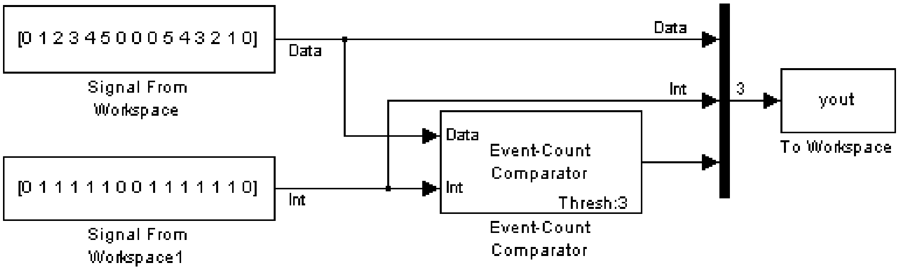
The Event-Count Comparator block records the number of nonzero inputs to the Data port during the period that the block is enabled by a high signal (the value 1) at the Int port. Both inputs must be scalars, and the Int input must be sample based. If the input to the Data port is frame based, the output is frame based; otherwise, the output is sample based.

When the number of accumulated nonzero inputs first equals the **Event threshold** setting, the block waits one additional sample interval, and then sets the output high (1). The block holds the output high until recording is restarted by a low-to-high (0-to-1) transition at the Int port.

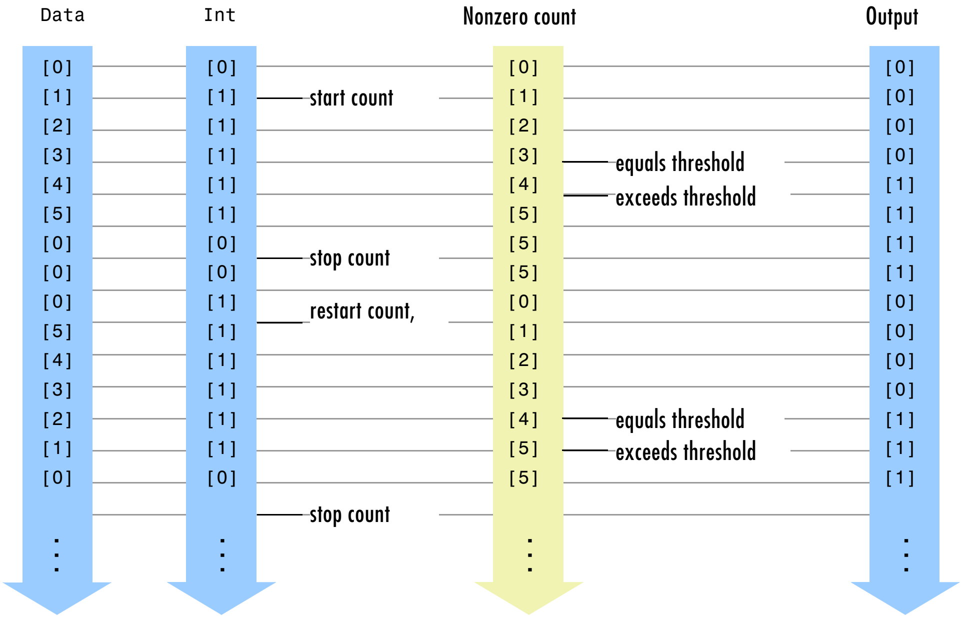
The Event-Count Comparator block accepts real and complex floating-point and fixed-point inputs.

Examples

In the model below, the Event-Count Comparator block (**Event threshold** = 3) detects two threshold crossings in the input to the Data port, one at sample 4 and one at sample 12.

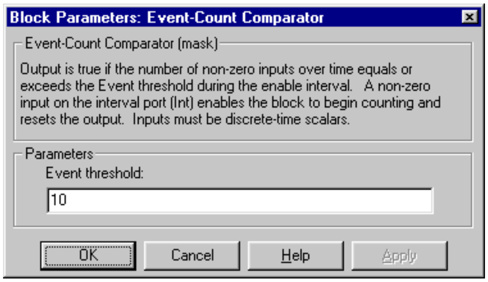


All inputs and outputs are multiplexed into the workspace variable yout, whose contents are shown in the figure below. The two left columns in the illustration show the inputs to the Data and Int ports, the center column shows the state of the block's internal counter, and the right column shows the block's output.



Data	Int		Nonzero count		Output
[0]	[0]		[0]		[0]
[1]	[1]	start count	[1]		[0]
[2]	[1]		[2]		[0]
[3]	[1]		[3]	equals threshold	[0]
[4]	[1]		[4]	exceeds threshold	[1]
[5]	[1]		[5]		[1]
[0]	[0]	stop count	[5]		[1]
[0]	[0]		[5]		[1]
[0]	[1]	restart count,	[0]		[0]
[5]	[1]		[1]		[0]
[4]	[1]		[2]		[0]
[3]	[1]		[3]		[0]
[2]	[1]		[4]	equals threshold	[1]
[1]	[1]		[5]	exceeds threshold	[1]
[0]	[0]		[5]		[1]
		stop count			
⋮	⋮		⋮		⋮

Dialog Box



Block Parameters: Event-Count Comparator

Event-Count Comparator (mask)

Output is true if the number of non-zero inputs over time equals or exceeds the Event threshold during the enable interval. A non-zero input on the interval port (Int) enables the block to begin counting and resets the output. Inputs must be discrete-time scalars.

Parameters

Event threshold:

10

Event threshold

Specify the value against which to compare the number of nonzero inputs. Tunable.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point
- Custom data types

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

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

See Also

Counter	DSP Blockset
Edge Detector	DSP Blockset

Extract Diagonal

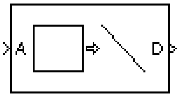
Purpose

Extract the main diagonal of the input matrix

Library

Math Functions / Matrices and Linear Algebra / Matrix Operations

Description

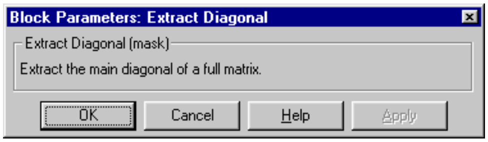


The Extract Diagonal block populates the 1-D output vector with the elements on the main diagonal of the M-by-N input matrix A.

```
D = diag(A)        Equivalent MATLAB code
```

The output vector has length `min(M,N)`, and is always sample based.

Dialog Box



Block Parameters: Extract Diagonal

Extract Diagonal (mask)

Extract the main diagonal of a full matrix.

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

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

See Also

Constant Diagonal Matrix	DSP Blockset
Create Diagonal Matrix	DSP Blockset
Extract Triangular Matrix	DSP Blockset
`diag`	MATLAB

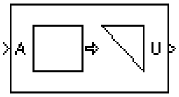
Purpose

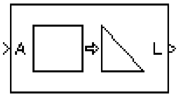
Extract the lower or upper triangle from an input matrix

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.

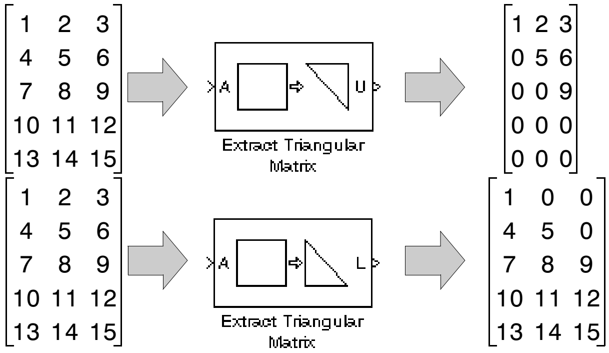
The **Extract** parameter selects between the two components of the input:

- `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.

Examples

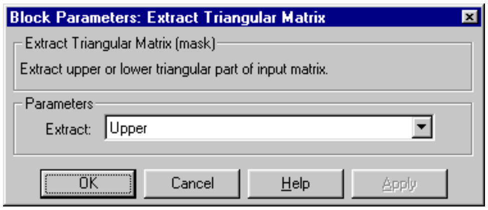
The example below shows the extraction of upper and lower triangles from a 5-by-3 input matrix.



$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \\ 13 & 14 & 15 \end{bmatrix} \qquad \begin{bmatrix} 1 & 2 & 3 \\ 0 & 5 & 6 \\ 0 & 0 & 9 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \\ 13 & 14 & 15 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 & 0 \\ 4 & 5 & 0 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \\ 13 & 14 & 15 \end{bmatrix}$$

Dialog Box



Block Parameters: Extract Triangular Matrix

Extract Triangular Matrix (mask)

Extract upper or lower triangular part of input matrix.

Parameters

Extract: Upper

Extract

The component of the matrix to copy to the output, upper triangle or lower triangle. 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 your data types to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-2.

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

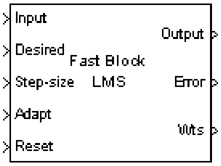
Purpose

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

Library

Filtering / Adaptive Filters

Description



Fast Block LMS Filter

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

The block calculates the filter weights using the Block LMS Filter equations. For more information, see “Block LMS Filter” on page 7-46. The Fast Block LMS Filter block implements the convolution operation involved in the calculations of the filtered output, y , and the weight update function in the frequency domain using the FFT algorithm used in the Overlap-Save FFT Filter block. See “Overlap-Save FFT Filter” on page 7-571 for more information.

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

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

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

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

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

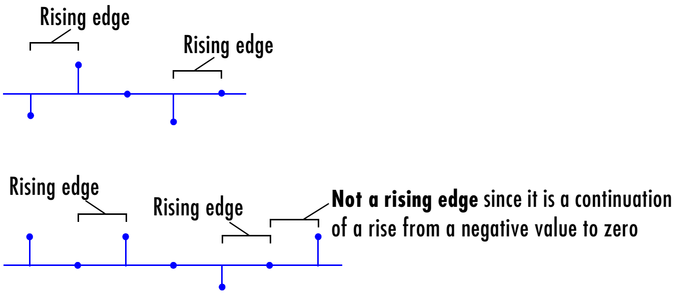
Enter the initial filter weights, $\hat{\mathbf{w}}(0)$, as a vector or a scalar in the **Initial value of filter weights** text box. If you enter a scalar, the block uses the scalar value to create a vector of filter weights. This vector has length equal to the filter length and all of its values are equal to the scalar value.

If you select the **Enable/disable adaptation via input port** check box, an Adapt port appears on the block. When the input to this port is nonzero, the block continuously updates the filter weights. When the input to this port is zero, the filter weights remain at their current values.

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

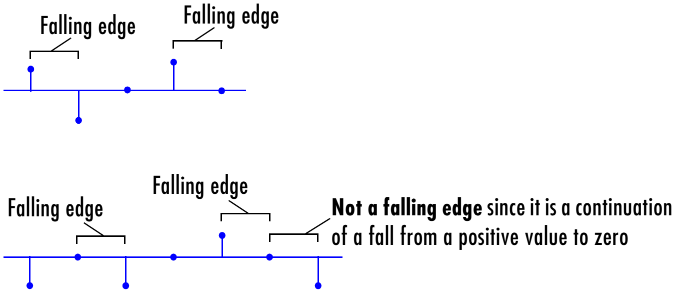
From the **Reset input** list, select `None` to disable the Reset port. To enable the Reset port, select one of the following from the **Reset input** list:

- `Rising edge` — Triggers a reset operation when the Reset input does one of the following:
  - Rises from a negative value to a positive value or zero
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)


Rising edge
Rising edge
Rising edge
Rising edge
Not a rising edge since it is a continuation of a rise from a negative value to zero

- `Falling edge` — Triggers a reset operation when the Reset input does one of the following:
  - Falls from a positive value to a negative value or zero

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


Falling edge
Falling edge
Falling edge
Falling edge
Not a falling edge since it is a continuation of a fall from a positive value to zero

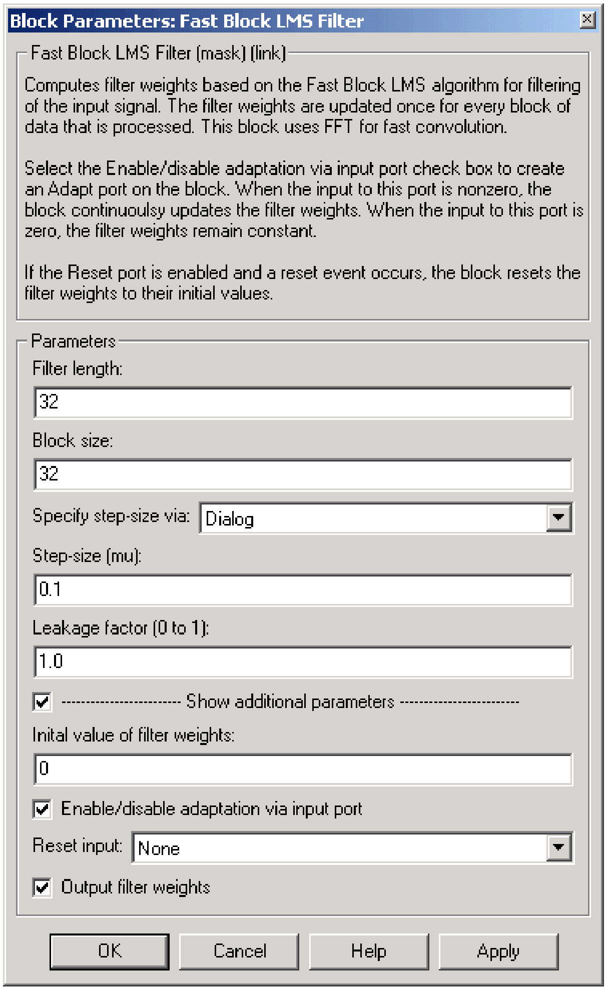
- `Either edge` — Triggers a reset operation when the Reset input is a `Rising edge` or `Falling edge` (as described above)
- `Non-zero sample` — Triggers a reset operation at each sample time that the Reset input is not zero

---

Note When running simulations in the Simulink `MultiTasking` mode, sample-based reset signals have a one-sample latency, and frame-based reset signals have one frame of latency. Thus, there is a one-sample or one-frame delay between the time the block detects a reset event, and when it applies the reset. For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" on page 2-98 and the topic called "The Simulation Parameters Dialog Box" in the Simulink documentation.

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

Dialog Box



Block Parameters: Fast Block LMS Filter

Fast Block LMS Filter (mask) (link)

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

Select the Enable/disable adaptation via input port check box to create an Adapt port on the block. When the input to this port is nonzero, the block continuoulsy updates the filter weights. When the input to this port is zero, the filter weights remain constant.

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

Parameters

Filter length: 32

Block size: 32

Specify step-size via: Dialog

Step-size (mu): 0.1

Leakage factor (0 to 1): 1.0

☑ ---------------------- Show additional parameters ----------------------

Inital value of filter weights: 0

☑ Enable/disable adaptation via input port

Reset input: None

☑ Output filter weights

Filter length

Enter the length of the FIR filter weights vector. The sum of the block size and the filter length must be a power of 2.

Block size

Enter the number of samples to acquire before the filter weights are updated. The input frame length must be an integer multiple of the block size. The sum of the block size and the filter length must be a power of 2.

Specify step-size via

Select `Dialog` to enter a value for mu, or select `Input port` to specify mu using the Step-size input port.

Step-size (mu)

Enter the step-size. Tunable.

Leakage factor (0 to 1)

Enter the leakage factor, $0 < 1 - \mu\alpha \leq 1$. Tunable.

Initial value of filter weights

Specify the initial values of the FIR filter weights.

Enable/disable adaptation via input port

Select this check box to enable the Adapt input port.

Reset input

Select this check box to enable the Reset input port.

Output filter weights

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

References

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

Supported Data Types

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

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

See Also

Kalman Adaptive Filter	DSP Blockset
LMS Filter	DSP Blockset
RLS Filter	DSP Blockset
Fast Block LMS Filter	DSP Blockset
Overlap-Save FFT Filter	DSP Blockset

See “Adaptive Filters” on page 3-46 for related information.

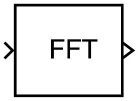
Purpose

Compute the FFT of the input

Library

Transforms

Description



The FFT block computes the fast Fourier transform (FFT) of each channel of an M-by-N or length-M input, u, where M must be a power of two. To work with other input sizes, use the Zero Pad block to pad or truncate the length-M dimension to a power-of-two length.

The output of the FFT block is equivalent to the MATLAB `fft` function:

```
y = fft(u,M)        % Equivalent MATLAB code
```

The *k*th entry of the *l*th output channel, $y(k, l)$, is equal to the *k*th point of the M-point discrete Fourier transform (DFT) of the *l*th input channel:

$$y(k,l) = \sum_{m=1}^{M} u(m,l)e^{-j2\pi(m-1)(k-1)/M} \qquad k = 1, \ldots, M$$

This block supports real and complex floating-point and fixed-point inputs.

Sections of This Reference Page

For information on block output characteristics and how to configure the block computation methods, see other sections of this reference page:

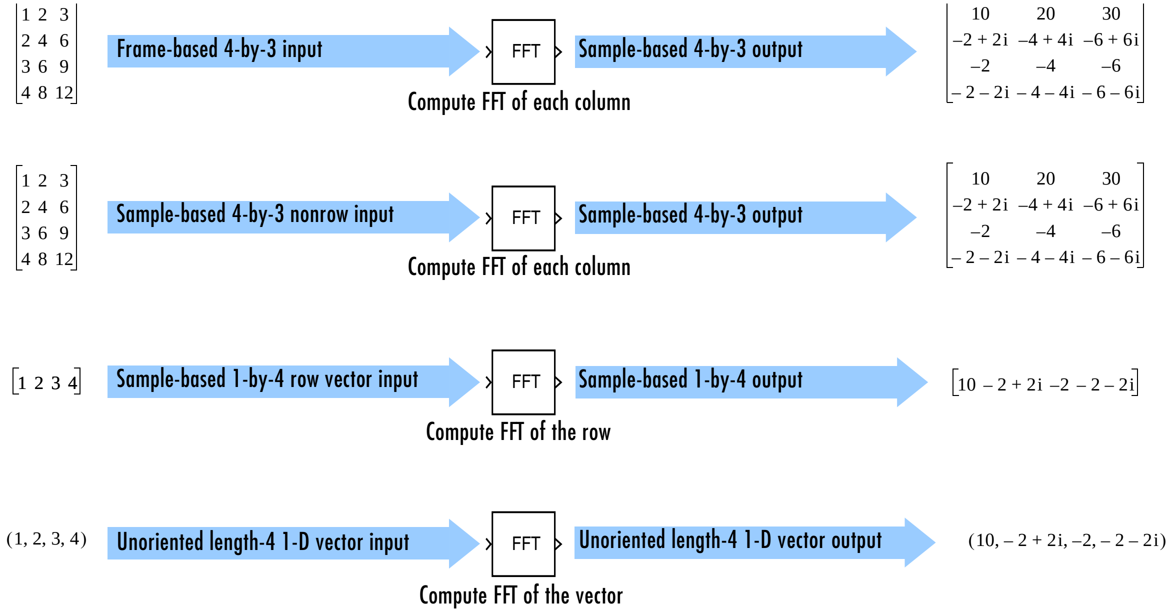
- “Input and Output Characteristics” on page 7-301
- “Selecting the Twiddle Factor Computation Method” on page 7-303
- “Optimizing the Table of Trigonometric Values” on page 7-304
- “Ordering Output Column Entries” on page 7-304
- “Algorithms Used for FFT Computation” on page 7-306
- “Examples” on page 7-309
- “Dialog Box” on page 7-310
- “Supported Data Types” on page 7-315
- “See Also” on page 7-316

Input and Output Characteristics

The following table describes valid inputs to the FFT block, their corresponding outputs, and the dimension along which the block computes the DFT.

Valid Block Inputs • Real- or complex-valued • Must be in linear order • M must be a power of two	**Dimension Along Which Block Computes DFT**	**Corresponding Block Output Characteristics** Output port rate = input port rate
Frame-based M-by-N matrix	Column	• Sample based • Complex valued • 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.
Sample-based M-by-N matrix, $M \neq 1$	Column	
Sample-based 1-by-M *row vector*	Row	
Unoriented length-M 1-D vector	Vector	Unoriented, length-M, complex-valued 1-D vector containing M-point DFT of input in linear or bit-reversed order

The following diagram shows the effects of the input size, dimension, and frame status on the output of the FFT block (see also `fft_ins_outs.mdl`).


[1 2 3; 2 4 6; 3 6 9; 4 8 12]
Frame-based 4-by-3 input
FFT
Sample-based 4-by-3 output
[10 20 30; −2 + 2i −4 + 4i −6 + 6i; −2 −4 −6; − 2 − 2i − 4 − 4i − 6 − 6i]
Compute FFT of each column
[1 2 3; 2 4 6; 3 6 9; 4 8 12]
Sample-based 4-by-3 nonrow input
FFT
Sample-based 4-by-3 output
[10 20 30; −2 + 2i −4 + 4i −6 + 6i; −2 −4 −6; − 2 − 2i − 4 − 4i − 6 − 6i]
Compute FFT of each column
[1 2 3 4]
Sample-based 1-by-4 row vector input
FFT
Sample-based 1-by-4 output
[10 − 2 + 2i −2 − 2 − 2i]
Compute FFT of the row
(1, 2, 3, 4)
Unoriented length-4 1-D vector input
FFT
Unoriented length-4 1-D vector output
(10, − 2 + 2i, −2, − 2 − 2i)
Compute FFT of the vector

Selecting the Twiddle Factor Computation Method

The **Twiddle factor computation** parameter determines how the block computes the necessary sine and cosine terms to calculate the term $e^{-j2\pi(m-1)(k-1)/M}$, shown in the first equation of this block reference page. This parameter has two settings, each with its advantages and disadvantages, as described in the following table. Note that only `Table lookup` mode is supported for fixed-point signals.

Twiddle Factor Computation Parameter Setting	Sine and Cosine Computation Method	Effect on Block Performance
`Table lookup`	The block computes and stores the trigonometric values before the simulation starts, and retrieves them during the simulation. When you generate code from the block, the processor running the generated code stores the trigonometric values computed by the block, and retrieves the values during code execution.	The block usually runs much more quickly, but requires extra memory for storing the precomputed trigonometric values. You can optimize the table for memory consumption or speed, as described in "Optimizing the Table of Trigonometric Values" 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 also set the **Optimize table for** parameter. This parameter optimizes the table of trigonometric values for speed or memory by varying the number of table entries as summarized in the following table.

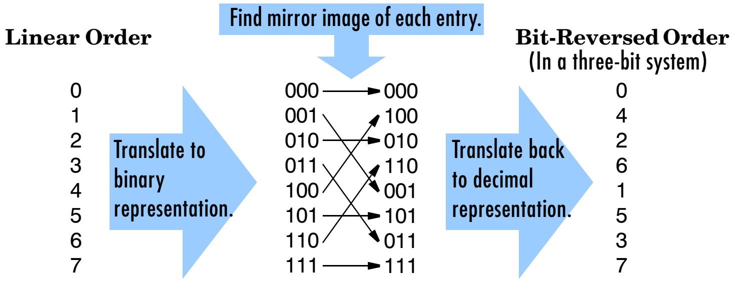
Optimize Table for Parameter Setting	Number of Table Entries for N-Point FFT	Memory Required for Single-Precision 512-Point FFT
`Speed`	$3N/4$ — floating point N — fixed point	1536 bytes
`Memory`	$N/4 + 1$ — floating point Not supported for fixed point	516 bytes

Ordering Output Column Entries

You can set the **Output in bit-reversed order** parameter to specify the ordering of the column elements of the output as either linear or bit-reversed.

Two numbers are bit-reversed values of each other when the binary representation of one is the mirror image of the binary representation of the other. For example, in a three-bit system, one and four are bit-reversed values of each other, since the three-bit binary representation of one, 001, is the mirror image of the three-bit binary representation of four, 100.

In the diagram below, the 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 finally translating the sequence back to its decimal representation. The resulting sequence is the original linearly ordered sequence in bit-reversed order.

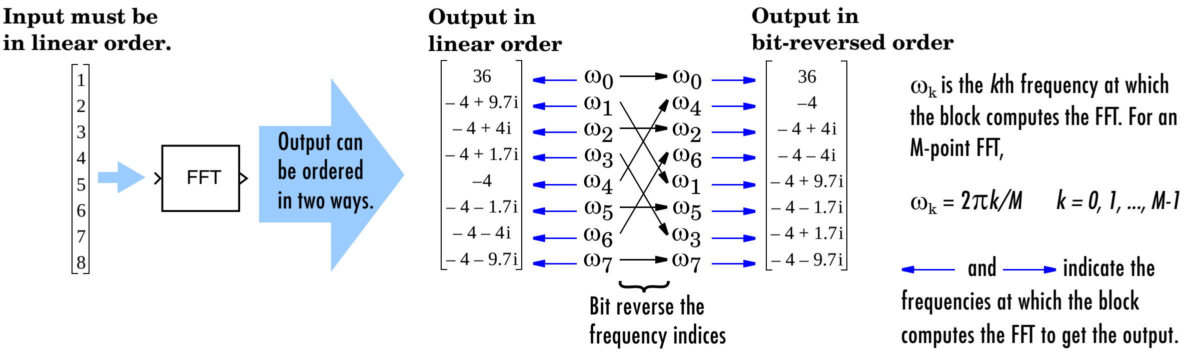


Set the **Output in bit-reversed order** parameter as follows to indicate the ordering of the output's column elements.

Parameter Setting	Ordering of Output Channel Elements	Output Column Entries
 □ Output in bit-reversed order	Linear order	*k*th column element is the DFT of the corresponding input column at the *k*th frequency.
 ☑ Output in bit-reversed order	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, "The Use of Bit-Reversed Outputs" on page 7-309.

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.


Input must be in linear order.
1
2
3
4
5
6
7
8
FFT
Output can be ordered in two ways.
Output in linear order
36
– 4 + 9.7i
– 4 + 4i
– 4 + 1.7i
–4
– 4 – 1.7i
– 4 – 4i
– 4 – 9.7i
ω_0 ω_1 ω_2 ω_3 ω_4 ω_5 ω_6 ω_7
ω_0 ω_4 ω_2 ω_6 ω_1 ω_5 ω_3 ω_7
Output in bit-reversed order
36
–4
– 4 + 4i
– 4 – 4i
– 4 + 9.7i
– 4 – 1.7i
– 4 + 1.7i
– 4 – 9.7i
Bit reverse the frequency indices
ω_k is the *k*th frequency at which the block computes the FFT. For an M-point FFT,
$\omega_k = 2\pi k/M \qquad k = 0, 1, ..., M-1$
← and → indicate the frequencies at which the block computes the FFT to get the output.

Algorithms Used for FFT Computation

Depending on whether the block input is floating-point or fixed-point, real- or complex-valued, and whether you want the output in linear or bit-reversed order, the block uses one or more of the following algorithms as summarized in the following tables:

- Radix-2 decimation-in-time (DIT) algorithm
- Radix-2 decimation-in-frequency (DIF) algorithm
- Half-length algorithm
- Double-signal algorithm

For floating-point signals:

Complexity of Input	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

For fixed-point signals:

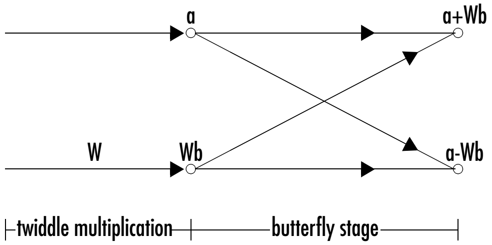
Complexity of Input	Output Ordering	Algorithms Used for FFT Computation
Real or complex	Linear	Radix-2 DIT
Real or complex	Bit-reversed	Radix-2 DIF

Fixed-Point Data Types

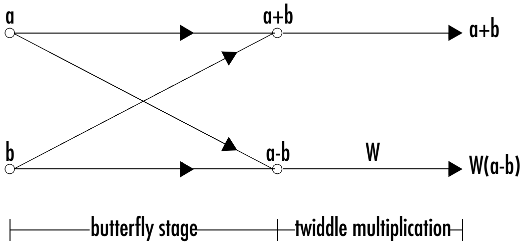
The diagrams below show the data types used within the FFT block for fixed-point signals. You can set the sine table, accumulator, product output, and output data types displayed in the diagrams in the FFT block mask as discussed in "Dialog Box" on page 7-310.

Inputs to the FFT block are first cast to the output data type and stored in the output buffer. Each butterfly stage then processes signals in the accumulator data type, with the final output of the butterfly being cast back into the output data type. A twiddle factor is multiplied in before each butterfly stage in a decimation-in-time FFT, and after each butterfly stage in a decimation-in-frequency FFT.

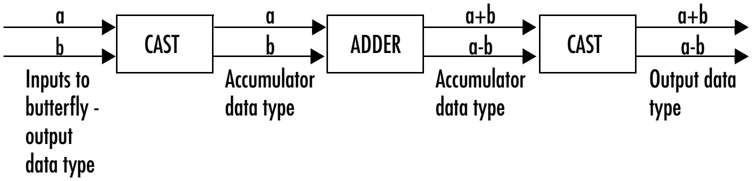
Decimation-in-time FFT



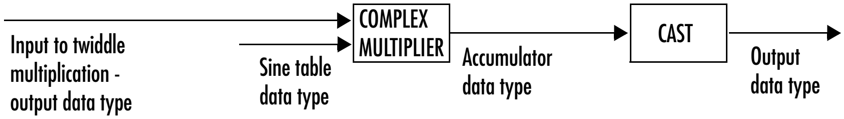
Decimation-in-frequency FFT



Butterfly stage data types



Twiddle multiplication data types



The output of the multiplier is in the accumulator data type since both of the inputs to the multiplier are complex. For details on the complex multiplication performed, refer to “Multiplication Data Types” on page 6-15.

Examples

The Use of Bit-Reversed Outputs

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, which 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 that is configured to accept input in bit-reversed order.

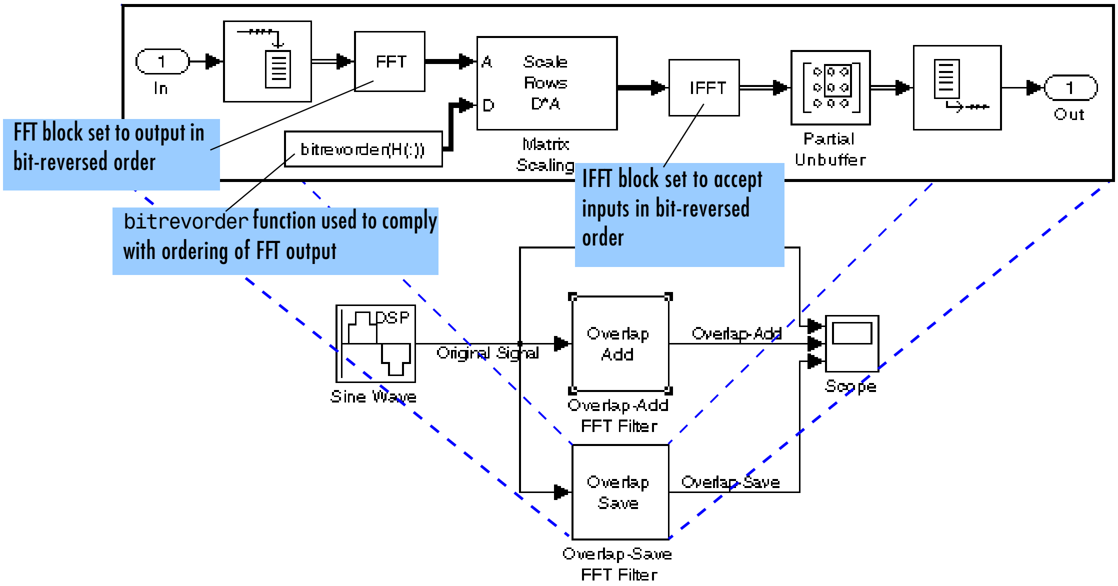
The following model shows the implementation of the Overlap-Save FFT Filter block. The implementation uses the FFT block in conjunction with an IFFT block, so the FFT block is set to output in bit-reversed order, and the IFFT block is set to accept inputs in bit-reversed order. Note that the implementation uses the `bitrevorder` function to put the vector H into bit-reversed order before multiplying it with the bit-reversed FFT outputs:

1. To open the demo model, type the following command at the MATLAB command line.

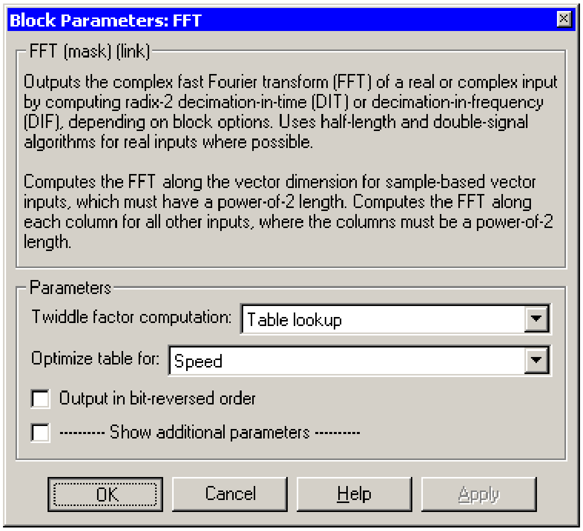
   ```
   olapfilt
   ```

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



FFT (mask) (link)

Outputs the complex fast Fourier transform (FFT) of a real or complex input by computing radix-2 decimation-in-time (DIT) or decimation-in-frequency (DIF), depending on block options. Uses half-length and double-signal algorithms for real inputs where possible.

Computes the FFT along the vector dimension for sample-based vector inputs, which must have a power-of-2 length. Computes the FFT along each column for all other inputs, where the columns must be a power-of-2 length.

Parameters

Twiddle factor computation: Table lookup

Optimize table for: Speed

□ Output in bit-reversed order

□ --------- Show additional parameters ---------

Twiddle factor computation

Specify the computation method of the term $e^{-j2\pi(m-1)(k-1)/M}$, shown in the first equation of this block reference page. In `Table lookup` mode, the block computes and stores the sine and cosine values before the simulation starts. In `Trigonometric fcn` mode, the block computes the sine and cosine values during the simulation. See "Selecting the Twiddle Factor Computation Method" on page 7-303.

This parameter must be set to `Table lookup` for fixed-point signals.

Optimize table for

Select the optimization of the table of sine and cosine values for `Speed` or `Memory`. This parameter is only available when the **Twiddle factor computation** parameter is set to `Table lookup`. See "Selecting the Twiddle Factor Computation Method" on page 7-303.

This parameter must be set to `Speed` for fixed-point signals.

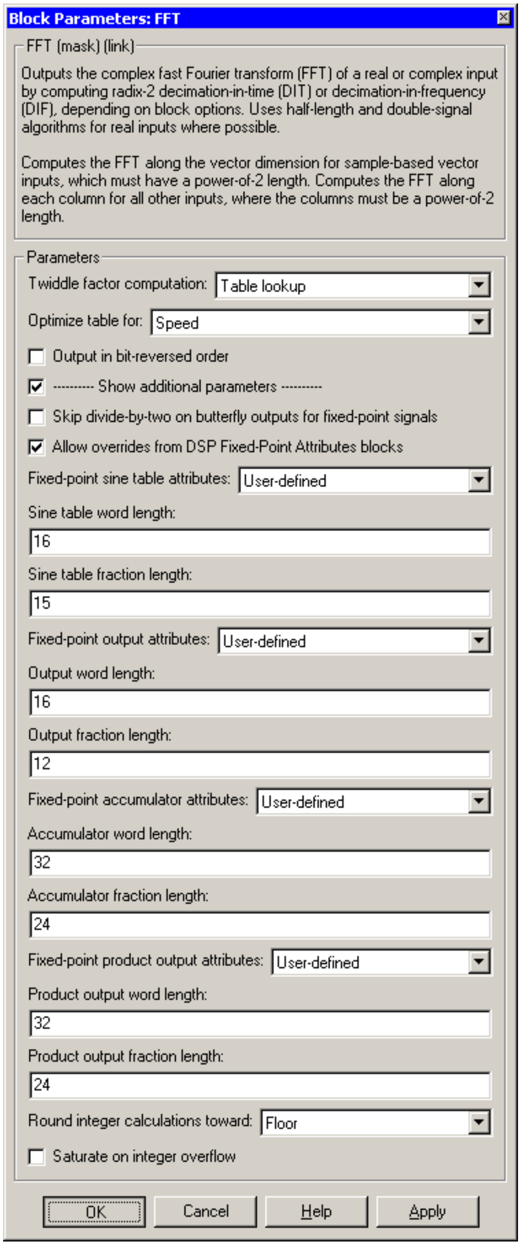
Output in bit-reversed order

Designate the order of the output channel elements relative to the ordering of the input elements. When selected, the output channel elements are in bit-reversed order relative to the input ordering. Otherwise, the output column elements are linearly ordered relative to the input ordering.

Linearly ordering the output requires extra data sorting manipulation, so in some situations it may be better to output in bit-reversed order as illustrated in the example, "The Use of Bit-Reversed Outputs" on page 7-309.

Show additional parameters

If selected, additional parameters specific to implementation of the block become visible as shown.



Block Parameters: FFT

FFT (mask) (link)

Outputs the complex fast Fourier transform (FFT) of a real or complex input by computing radix-2 decimation-in-time (DIT) or decimation-in-frequency (DIF), depending on block options. Uses half-length and double-signal algorithms for real inputs where possible.

Computes the FFT along the vector dimension for sample-based vector inputs, which must have a power-of-2 length. Computes the FFT along each column for all other inputs, where the columns must be a power-of-2 length.

Parameters

Twiddle factor computation: Table lookup

Optimize table for: Speed

□ Output in bit-reversed order

☑ --------- Show additional parameters ---------

□ Skip divide-by-two on butterfly outputs for fixed-point signals

☑ Allow overrides from DSP Fixed-Point Attributes blocks

Fixed-point sine table attributes: User-defined

Sine table word length: 16

Sine table fraction length: 15

Fixed-point output attributes: User-defined

Output word length: 16

Output fraction length: 12

Fixed-point accumulator attributes: User-defined

Accumulator word length: 32

Accumulator fraction length: 24

Fixed-point product output attributes: User-defined

Product output word length: 32

Product output fraction length: 24

Round integer calculations toward: Floor

□ Saturate on integer overflow

Skip divide-by-2 on butterfly outputs for fixed-point signals

When this parameter is selected, no scaling occurs. When this parameter is not selected, the output of each butterfly of the FFT is divided by two for fixed-point signals.

Allow overrides from DSP Fixed-Point Attributes blocks

If you select this parameter, fixed-point data types for this block may be set by DSP Fixed-Point Attributes blocks in your model. If this parameter is unselected, the data types are always set by the parameters in the block mask.

Fixed-point sine table attributes

Choose how you will specify the word length and fraction length of the values of the sine table. If you select `Same as input`, the word and fraction lengths of the sine table values are the same as those of the input of the block. If you select `Same as output`, they are the same as those of the output of the block. If you select `User-defined`, the **Fixed-point sine table word length** and **Fixed-point sine table fraction length** parameters become visible.

The sine table values do not obey the **Round integer calculations toward** and **Saturate on integer overflow** parameters; they are always saturated and rounded to `Nearest`.

Fixed-point sine table word length

Specify the word length, in bits, of the sine table values. This parameter is only visible if `User-defined` is specified for the **Fixed-point sine table attributes** parameter.

Fixed-point sine table fraction length

Specify the fraction length, in bits, of the sine table values. This parameter is only visible if `User-defined` is specified for the **Fixed-point sine table attributes** parameter.

Fixed-point output attributes

Choose how you will specify the output word length and fraction length. If you select `Same as input`, these characteristics will match those of the input to the block. If you select `User-defined`, the **Output word length** and **Output fraction length** parameters become visible.

Output word length

Specify the word length, in bits, of the output. This parameter is only visible if `User-defined` is specified for the **Fixed-point output attributes** parameter.

Output fraction length

Specify the fraction length, in bits, of the output. This parameter is only visible if `User-defined` is specified for the **Fixed-point output attributes** parameter.

Fixed-point accumulator attributes

Use this parameter to specify how you would like to designate the accumulator word and fraction lengths. Refer to "Fixed-Point Data Types" on page 7-32 and "Multiplication Data Types" on page 6-15 for illustrations depicting the use of the accumulator data type in this block.

If you select `Same as input`, the accumulator word and fraction lengths are the same as those of the input to the block. If you select `Same as output`, they are the same as those of the output of the block. If you select `User-defined`, the **Accumulator word length** and **Accumulator fraction length** parameters become visible.

Accumulator word length

Specify the word length, in bits, of the accumulator. This parameter is only visible when `User-defined` is specified for the **Fixed-point accumulator attributes** parameter.

Accumulator fraction length

Specify the fraction length, in bits, of the accumulator. This parameter is only visible when `User-defined` is specified for the **Fixed-point accumulator attributes** parameter.

Fixed-point product output attributes

Use this parameter to specify how you would like to designate the product output word and fraction lengths. Refer to "Fixed-Point Data Types" on page 7-32 and "Multiplication Data Types" on page 6-15 for illustrations depicting the use of the product output data type in this block.

If you select `Same as input`, `Same as output`, or `Same as accumulator`, the product output word and fraction lengths are the same as those of the input, output, or accumulator of the block, respectively. If you select `User-defined`, the **Product output word length** and **Product output fraction length** parameters become visible.

Product output word length

Specify the word length, in bits, of the product output. This parameter is only visible when `User-defined` is specified for the **Fixed-point product output attributes** parameter.

Product output fraction length

Specify the fraction length, in bits, of the product output. This parameter is only visible when `User-defined` is specified for the **Fixed-point product output attributes** parameter.

Round integer calculations toward

Select the rounding mode for fixed-point operations. The sine table values do not obey this parameter; they always round to `Nearest`.

Saturate on integer overflow

If selected, overflows saturate. The sine table values do not obey this parameter; they are always saturated.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point
- 8-, 16-, and 32-bit signed integers

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

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

Filter Realization Wizard

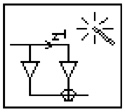
Purpose

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.

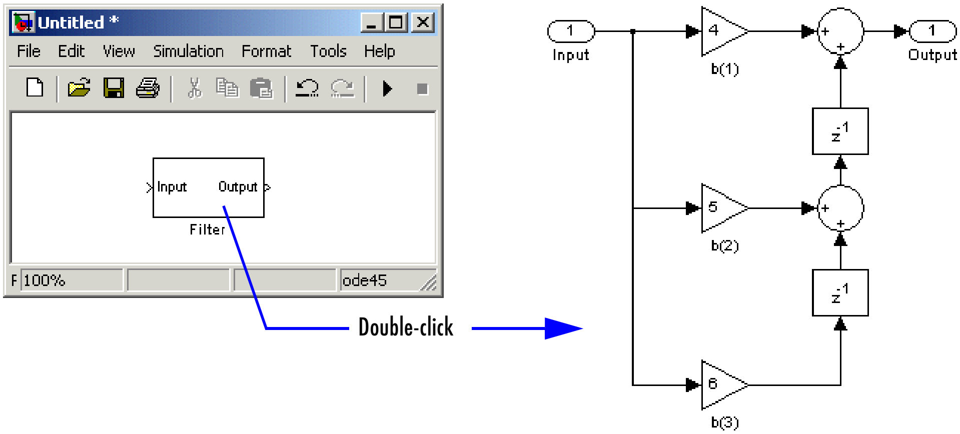
Steps to Implementing a Filter with This Block

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 7-321.
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 7-322.
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 7-324.
   - Select the filter structure optimizations. To learn more, see "Optimizing the Filter Structure" on page 7-325.
   - 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 7-326.

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 7-326.

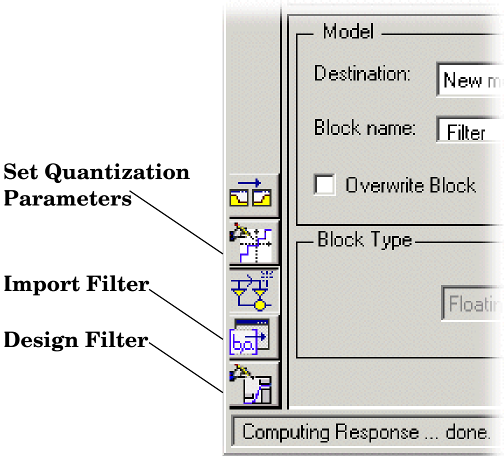
Note *Running* a model containing implementations of non-double-precision filters requires the Fixed-Point Blockset, but you can still edit models containing such filter implementations without the Fixed-Point Blockset. For more information, see the topic on licensing information in the Fixed-Point Blockset 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.

Click the buttons to open the corresponding FDATool panels.



Model

Destination: New m

Block name: Filter

□ Overwrite Block

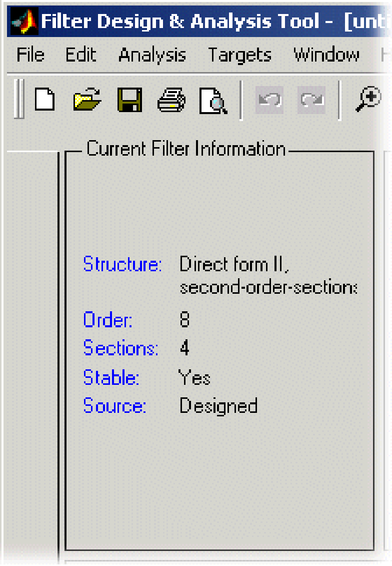
Block Type

Floati

Computing Response ... done.

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.


Filter Design & Analysis Tool - [unt
File Edit Analysis Targets Window

Current Filter Information

Structure: Direct form II, second-order-sections

Order: 8

Sections: 4

Stable: Yes

Source: Designed

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.

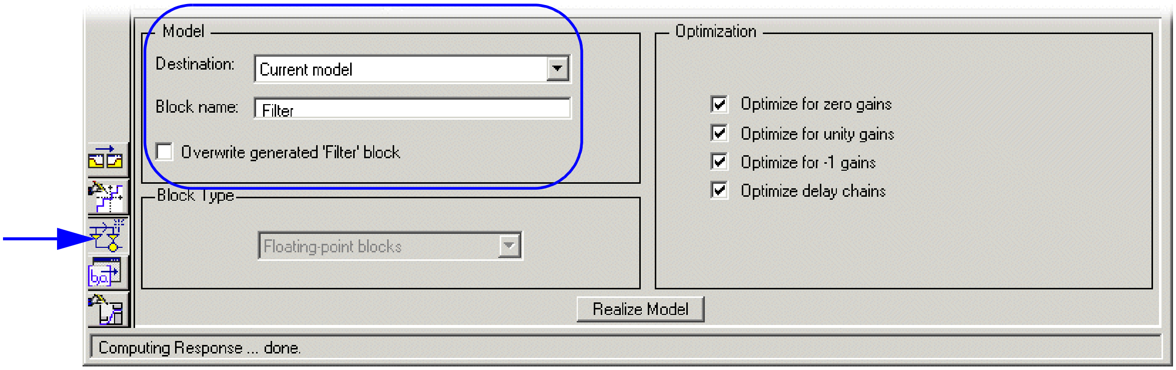
2 Set the **Destination** parameter to one of the following:

   - `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:

   - Select the **Overwrite generated 'Filter' block** check box — 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.
   - Clear the **Overwrite generated 'Filter' block** check box — 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`.



Model

Destination: Current model

Block name: Filter

☐ Overwrite generated 'Filter' block

Block Type

Floating-point blocks

Optimization

☑ Optimize for zero gains

☑ Optimize for unity gains

☑ Optimize for -1 gains

☑ Optimize delay chains

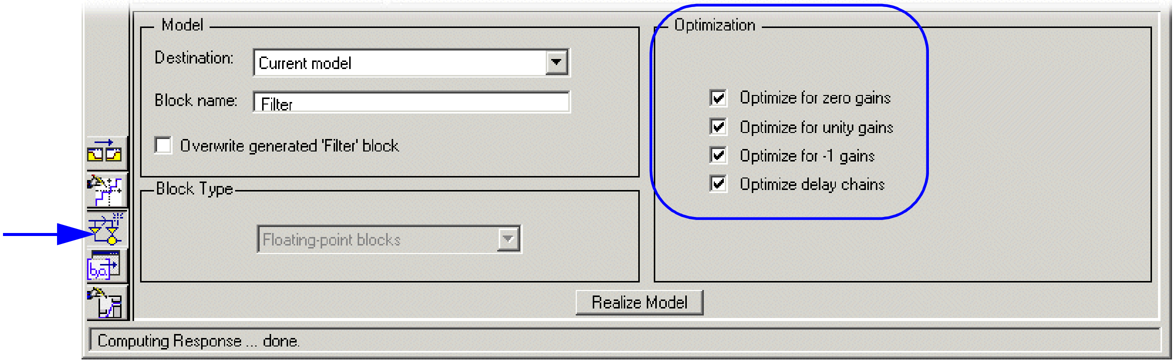
Computing Response ... done.

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.



Model

Destination: Current model

Block name: Filter

☐ Overwrite generated 'Filter' block

Block Type

Floating-point blocks

Optimization

☑ Optimize for zero gains

☑ Optimize for unity gains

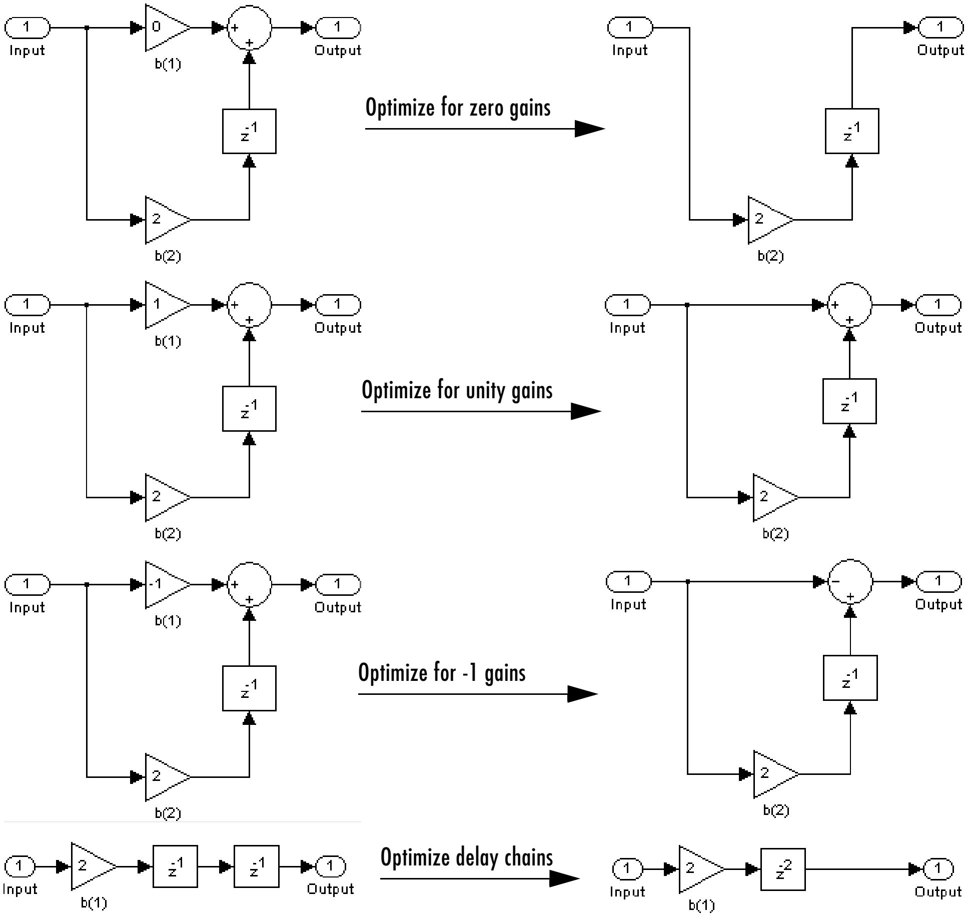
☑ Optimize for -1 gains

☑ Optimize delay chains

Computing Response ... done.

- **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*.

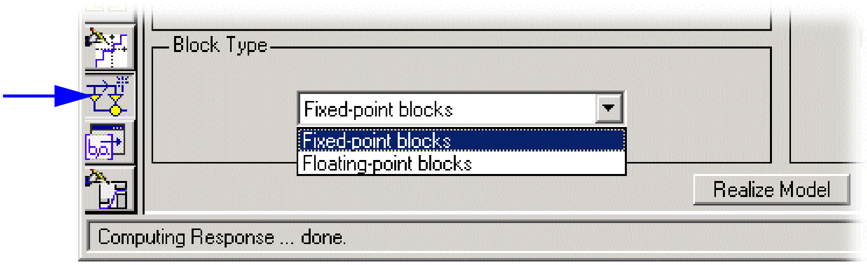


Setting the Data Type of the Filter Implementation

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 7-321.

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.



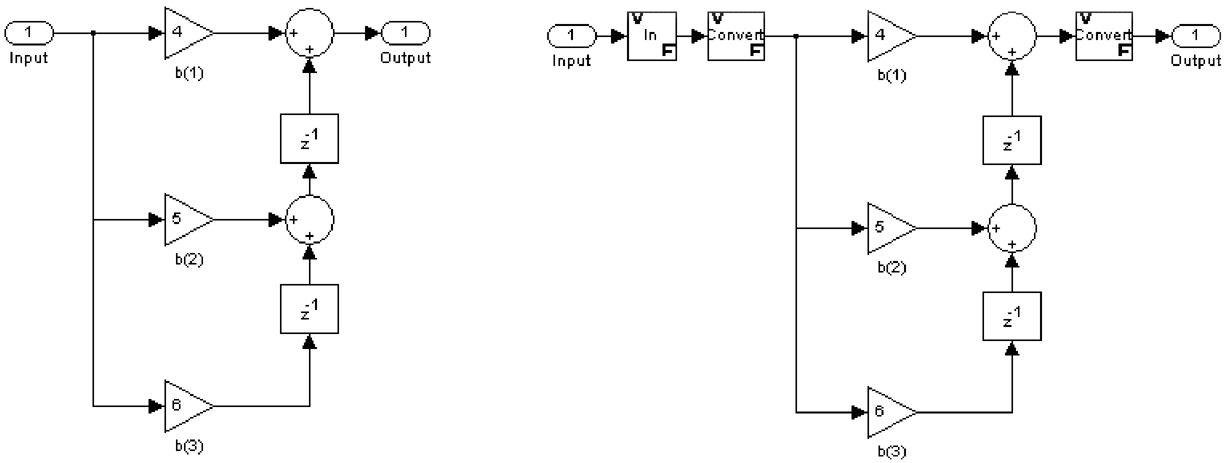
Block Type: Fixed-point blocks

Computing Response ... done.

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 7-321.	Sum, Gain, and Integer Delay blocks configured in the specified filter structure, and set to make double-precision computations. See the figure called Implementations of Double-Precision and Fixed-Point Filters.
Fixed-point blocks	When implementing any of the following as described in "Specifying the Filter and Its Data Type Support" on page 7-321: • 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 the figure called Implementations of Double-Precision and Fixed-Point Filters.

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.



Double-precision filter implemented with Sum, Gain, and Integer Delay blocks

Fixed-point filter implemented with Sum, Gain, Integer Delay, Gateway In, and Conversion blocks

Implementations of Double-Precision and Fixed-Point Filters

Corresponding Method for dfilt and qfilt

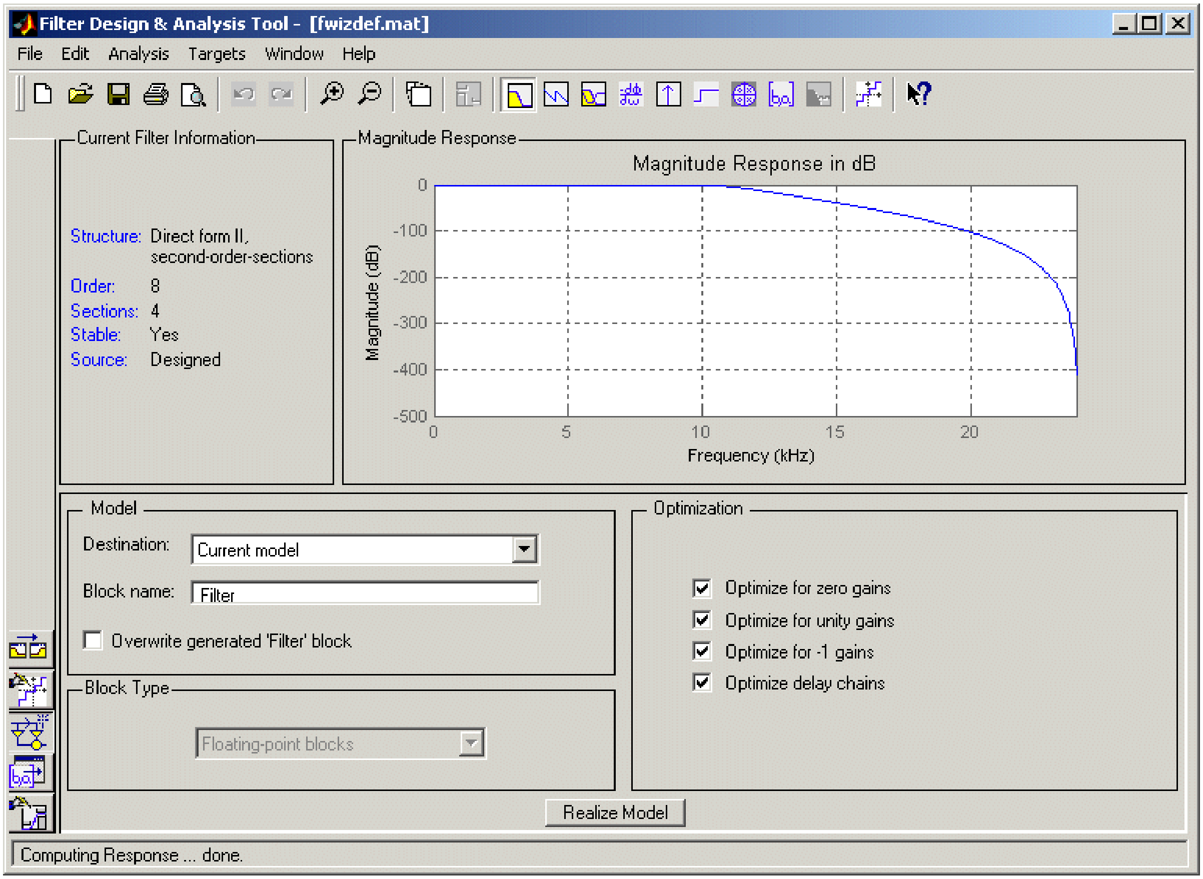
The `dfilt` (digital filter) object in Signal Processing Toolbox and the `qfilt` (quantized filter) object in Filter Design Toolbox both have a method, `realizemdl`, that allows you to access the capabilities of the Filter Realization Wizard from the command line.

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 7-321.



Current Filter Information

Structure: Direct form II, second-order-sections

Order: 8

Sections: 4

Stable: Yes

Source: Designed

Model

Destination: Current model

Block name: Filter

☐ Overwrite generated 'Filter' block

Block Type

Floating-point blocks

Optimization

☑ Optimize for zero gains

☑ Optimize for unity gains

☑ Optimize for -1 gains

☑ Optimize delay chains

Computing Response ... done.

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 7-324.

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 7-326.

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 7-325.

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 7-325.

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 7-325.

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 7-325.

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 7-321.

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

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

- Chapter 3, "Filters" — Examples of when and how to use DSP Blockset filtering blocks
- "Choosing Between Filter Design Blocks" on page 3-14

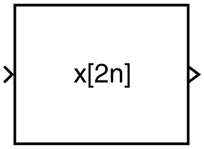
Purpose

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 FIR filter.
- The block downsamples the filtered data to a lower rate by discarding K-1 consecutive samples following every sample retained.

The FIR Decimation block implements the above FIR filtering and downsampling steps together using a polyphase filter structure, which is more efficient than straightforward filter-then-decimate algorithms. See N.J. Fliege, *Multirate Digital Signal Processing: Multirate Systems, Filter Banks, Wavelets* for more information.

The **FIR filter coefficients** parameter specifies the numerator coefficients of the FIR filter transfer function H(*z*).

$$H(z) = B(z) = b_1 + b_2 z^{-1} + \ldots + b_m z^{-(m-1)}$$

The length-*m* coefficient vector, `[b(1) b(2) ... b(m)]`, can be generated by one of the filter design functions in the Signal Processing Toolbox, such as the `fir1` function used in the example below. The filter should be lowpass with normalized cutoff frequency no greater than 1/K. All filter states are internally initialized to zero.

The FIR Decimation block supports real and complex floating-point inputs, and real fixed-point inputs. This block supports triggered subsystems if you select `Maintain input frame rate` for the **Framing** parameter.

Sample-Based Operation

An M-by-N sample-based matrix input is treated as M∗N independent channels, and the block decimates each channel over time. The output sample period is K times longer than the input sample period ($T_{so} = KT_{si}$), and the input and output sizes are identical.

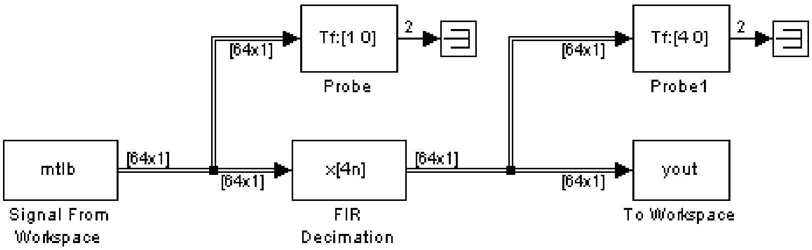
Frame-Based Operation

An M_i-by-N frame-based matrix input is treated as N independent channels, and the block decimates each channel over time. The **Framing** parameter determines how the block adjusts the rate at the output to accommodate the reduced number of samples. There are two available options:

- `Maintain input frame size`

  The block generates the output at the decimated rate by using a proportionally longer frame *period* at the output port than at the input port. For decimation by a factor of K, the output frame period is K times longer than the input frame period ($T_{fo} = KT_{fi}$), but the input and output frame sizes are equal.

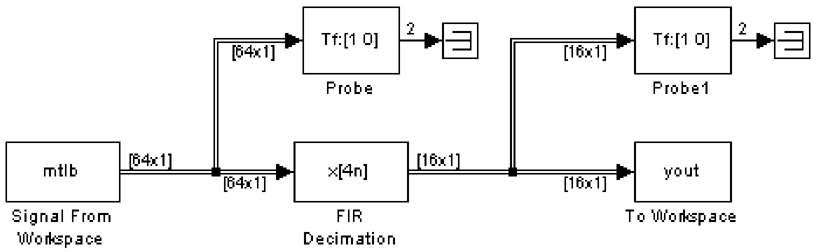
  The example below shows a single-channel input with a frame period of 1 second (**Sample time** = `1/64` and **Samples per frame** = `64` in the Signal From Workspace block) being decimated by a factor of 4 to a frame period of 4 seconds. The input and output frame sizes are identical.



- `Maintain input frame rate`

  The block generates the output at the decimated rate by using a proportionally smaller frame *size* than the input. For decimation by a factor of K, the output frame size is K times smaller than the input frame size ($M_o = M_i/K$), but the input and output frame rates are equal. The input frame size, M_i, must be a multiple of the decimation factor, K.

  The example below shows a single-channel input of frame size 64 being decimated by a factor of 4 to a frame size of 16. The block's input and output frame rates are identical.



Latency

Zero Latency. The FIR Decimation block has *zero tasking latency* for all single-rate operations. The block is single-rate for the particular combinations of sampling mode and parameter settings shown in the table below.

Sampling Mode	Parameter Settings
Sample based	**Decimation factor** parameter, K, is `1`.
Frame based	**Decimation factor** parameter, K, is `1`, *or* **Framing** parameter is `Maintain input frame rate`.

Note that in sample-based mode, single-rate operation occurs only in the trivial case of factor-of-1 decimation.

The block also has zero latency for sample-based multirate operations in the Simulink single-tasking mode. Zero tasking latency means that the block propagates the first filtered input sample (received at t=0) as the first output sample, followed by filtered input samples K+1, 2K+1, and so on.

Nonzero Latency. The FIR Decimation block is multirate for all settings other than those in the 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.

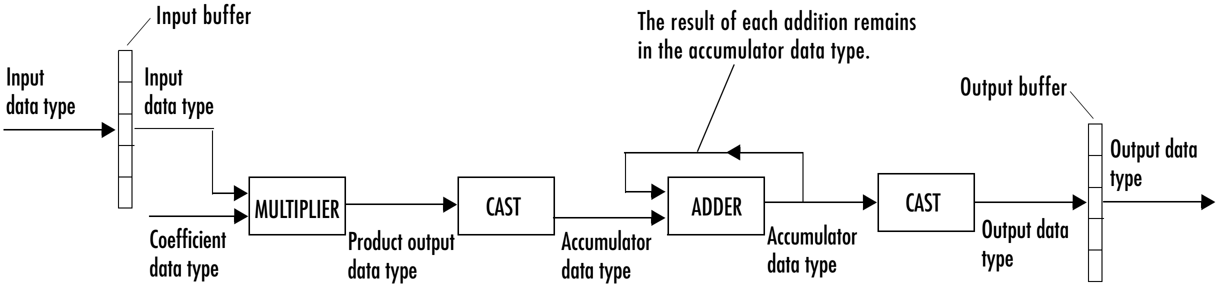
In cases of *one-frame latency*, the first M_i output rows contain zeros, where M_i is the input frame size. The first filtered input sample (first filtered row of the input matrix) appears in the output as sample M_i+1, followed by filtered input samples K+1, 2K+1, and so on. See the example below for an illustration of this case.

See "Excess Algorithmic Delay (Tasking Latency)" on page 2-98 and the topic called The Simulation Parameters Dialog Box in the Simulink documentation for more information about block rates and the Simulink tasking modes.

If the block exhibits latency, enter a value in the **Output buffer initial conditions** text box to specify the value to output at the output port until the first filtered input sample is available. The default initial condition value is 0.

Fixed-Point Data Types

The following diagram shows the data types used within the FIR Decimation block for fixed-point signals.

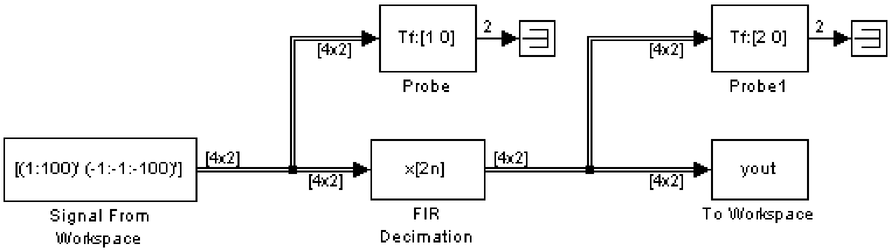


You can set the product output, accumulator, and output data types in the block mask as discussed in “Dialog Box” on page 7-339 below. The diagram shows that input data is stored in the input buffer in the same data type and scaling as the input. Filtered data is stored in the output buffer in the output data type and scaling that is set by you in the block mask. Any initial conditions are also stored in the output buffer in the output data type and scaling set by you in the block mask.

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 this is a frame-based multirate model, the first four (M_i) output rows are zero. The first filtered input matrix row appears in the output as sample 5 (that is, sample M_i+1).

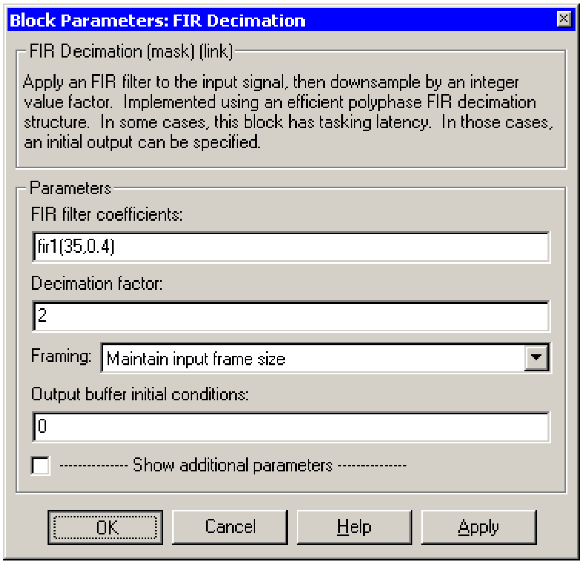
Example 2

The Polyphase FIR Decimation demo (`polyphaseDec_demo`) illustrates the underlying polyphase implementations of the FIR Decimation block. Run the demo and view the results on the scope. The output of the FIR Decimation block is the same as the output of the Polyphase Decimation Filter block.

Example 3

The `dspmrf_menu` demo illustrates the use of the FIR Decimation block in a number of multistage multirate filters.

Dialog Box



Block Parameters: FIR Decimation

FIR Decimation (mask) (link)

Apply an FIR filter to the input signal, then downsample by an integer value factor. Implemented using an efficient polyphase FIR decimation structure. In some cases, this block has tasking latency. In those cases, an initial output can be specified.

Parameters

FIR filter coefficients: fir1(35,0.4)

Decimation factor: 2

Framing: Maintain input frame size

Output buffer initial conditions: 0

□ -------------- Show additional parameters --------------

FIR filter coefficients

Specify the lowpass FIR filter coefficients, in descending powers of z.

Decimation factor

Specify the integer factor, K, by which to decrease the sample rate of the input sequence.

Framing

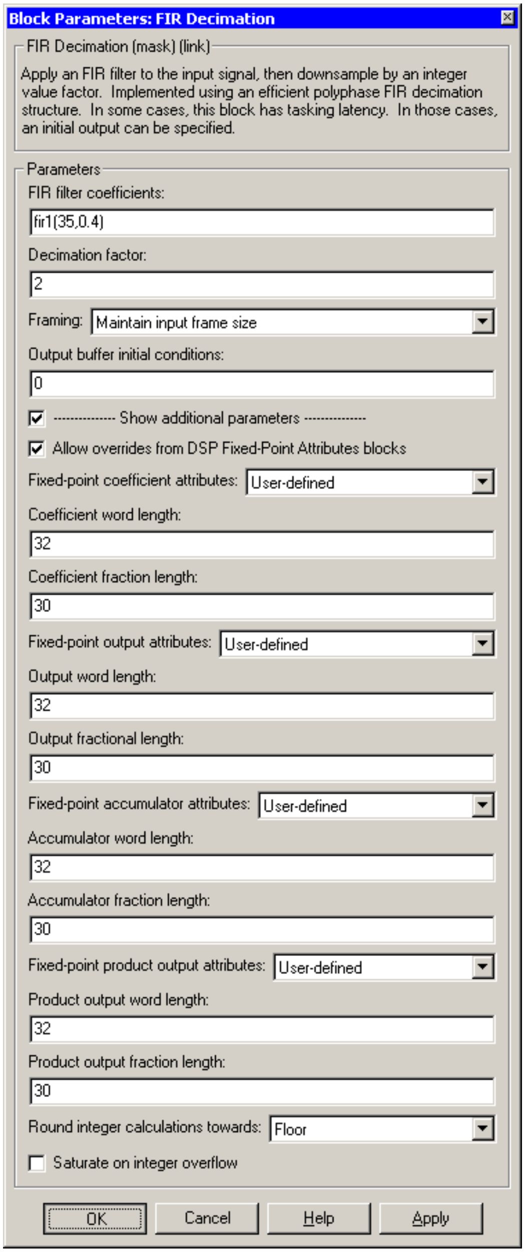
For frame-based operation, specify the method by which to implement the decimation; reduce the output frame rate, or reduce the output frame size. This parameter may not be set to `Maintain input frame rate` for sample-based signals.

Output buffer initial conditions

If the block exhibits latency, enter a value in the **Output buffer initial conditions** text box to specify the value to output at the output port until the first filtered input sample is available. The default initial condition value is zero.

Show additional parameters

If selected, additional parameters specific to implementation of the block become visible as shown.



Block Parameters: FIR Decimation

FIR Decimation (mask) (link)

Apply an FIR filter to the input signal, then downsample by an integer value factor. Implemented using an efficient polyphase FIR decimation structure. In some cases, this block has tasking latency. In those cases, an initial output can be specified.

Parameters

FIR filter coefficients: fir1(35,0.4)

Decimation factor: 2

Framing: Maintain input frame size

Output buffer initial conditions: 0

☑ ------------- Show additional parameters -------------

☑ Allow overrides from DSP Fixed-Point Attributes blocks

Fixed-point coefficient attributes: User-defined

Coefficient word length: 32

Coefficient fraction length: 30

Fixed-point output attributes: User-defined

Output word length: 32

Output fractional length: 30

Fixed-point accumulator attributes: User-defined

Accumulator word length: 32

Accumulator fraction length: 30

Fixed-point product output attributes: User-defined

Product output word length: 32

Product output fraction length: 30

Round integer calculations towards: Floor

☐ Saturate on integer overflow

Allow overrides from DSP Fixed-Point Attributes blocks

If you select this parameter, fixed-point data types for this block may be set by DSP Fixed-Point Attributes blocks in your model. If this parameter is unselected, the data types are always set by the parameters in the block mask.

Fixed-point coefficient attributes

Choose how you will specify the word length and fraction length of the filter coefficients. If you select `Same as input`, these characteristics will match those of the input to the block. If you select `User-defined`, the **Coefficient word length** and **Coefficient fraction length** parameters become visible.

The coefficients do not obey the **Round integer calculations toward** and the **Saturate on integer overflow** parameters; they are always saturated and rounded to `Nearest`.

Coefficient word length

Specify the word length, in bits, of the coefficients. This parameter is only visible if `User-defined` is specified for the **Fixed-point coefficient attributes** parameter.

Coefficient fraction length

Specify the fraction length, in bits, of the coefficients. This parameter is only visible if `User-defined` is specified for the **Fixed-point coefficient attributes** parameter.

Fixed-point output attributes

Choose how you will specify the output word length and fraction length. If you select `Same as input`, these characteristics will match those of the input to the block. If you select `User-defined`, the **Output word length** and **Output fraction length** parameters become visible.

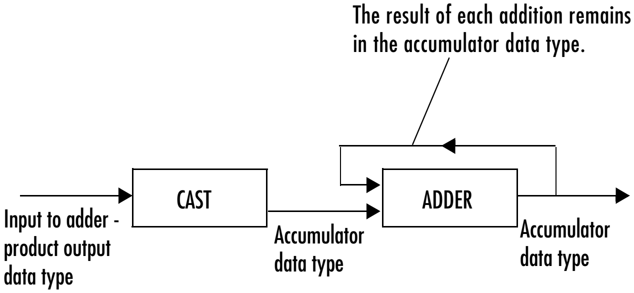
Output word length

Specify the word length, in bits, of the output. This parameter is only visible if `User-defined` is specified for the **Fixed-point output attributes** parameter.

Output fraction length

Specify the fraction length, in bits, of the output. This parameter is only visible if `User-defined` is specified for the **Fixed-point output attributes** parameter.

Fixed-point accumulator attributes


The result of each addition remains in the accumulator data type.
CAST
ADDER
Input to adder - product output data type
Accumulator data type
Accumulator data type

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

If you select `Same as output`, the accumulator word and fraction lengths are the same as those of the output of the block. If you select `User-defined`, the **Accumulator word length** and **Accumulator fraction length** parameters become visible.

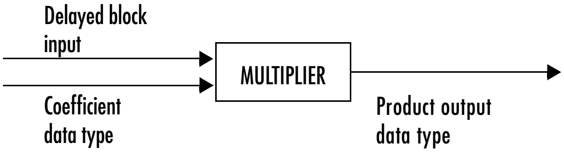
Accumulator word length

Specify the word length, in bits, of the accumulator. This parameter is only visible when `User-defined` is specified for the **Fixed-point accumulator attributes** parameter.

Accumulator fraction length

Specify the fraction length, in bits, of the accumulator. This parameter is only visible when `User-defined` is specified for the **Fixed-point accumulator attributes** parameter.

Fixed-point product output attributes



As depicted above, the output of the multiplier is placed into the product output data type and scaling. Use this parameter to specify how you would like to designate this product output word and fraction lengths.

If you select `Same as accumulator`, the product output word and fraction lengths are the same as those of the accumulator of the block. If you select `Same as output`, they are the same as those of the output of the block. If you select `User-defined`, the **Product output word length** and **Product output fraction length** parameters become visible.

Product output word length

Specify the word length, in bits, of the product output. This parameter is only visible when `User-defined` is specified for the **Fixed-point product output attributes** parameter.

Product output fraction length

Specify the fraction length, in bits, of the product output. This parameter is only visible when `User-defined` is specified for the **Fixed-point product output attributes** parameter.

Round integer calculations toward

Select the rounding mode for fixed-point operations. The filter coefficients do not obey this parameter; they always round to `Nearest`.

Saturate on integer overflow

If selected, overflows saturate. The filter coefficients do not obey this parameter; they are always saturated.

References

Fliege, N. J. *Multirate Digital Signal Processing: Multirate Systems, Filter Banks, Wavelets*. West Sussex, England: John Wiley & Sons, 1994.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point
- 8-, 16-, and 32-bit signed integers

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

See Also

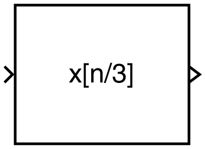
Downsample	DSP Blockset
FIR Interpolation	DSP Blockset
FIR Rate Conversion	DSP Blockset
`decimate`	Signal Processing Toolbox
`fir1`	Signal Processing Toolbox
`fir2`	Signal Processing Toolbox
`firls`	Signal Processing Toolbox
`remez`	Signal Processing Toolbox

FIR Interpolation

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 FIR filter.

The FIR Interpolation block implements the above upsampling and FIR filtering steps together using a polyphase filter structure, which is more efficient than straightforward upsample-then-filter algorithms. See N.J. Fliege, *Multirate Digital Signal Processing: Multirate Systems, Filter Banks, Wavelets* for more information.

The **FIR filter coefficients** parameter specifies the numerator coefficients of the FIR filter transfer function H(z).

$$H(z) = B(z) = b_1 + b_2 z^{-1} + \ldots + b_m z^{-(m-1)}$$

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.

The FIR Interpolation block supports real and complex floating-point inputs, and real fixed-point inputs. This block supports triggered subsystems if you select `Maintain input frame rate` for the **Framing** parameter.

Sample-Based Operation

An M-by-N sample-based matrix input is treated as M∗N independent channels, and the block interpolates each channel over time. The output sample period is L times shorter than the input sample period ($T_{so} = T_{si}/L$), and the input and output sizes are identical.

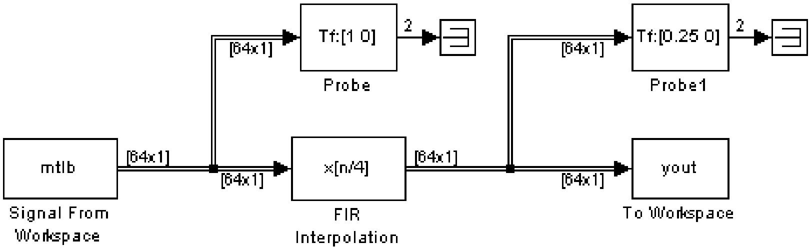
Frame-Based Operation

An M_i-by-N frame-based matrix input is treated as N independent channels, and the block decimates each channel over time. The **Framing** parameter determines how the block adjusts the rate at the output to accommodate the added samples. There are two available options:

- `Maintain input frame size`

  The block generates the output at the interpolated rate by using a proportionally shorter frame *period* at the output port than at the input port. For interpolation by a factor of L, the output frame period is L times shorter than the input frame period ($T_{fo} = T_{fi}/L$), but the input and output frame sizes are equal.

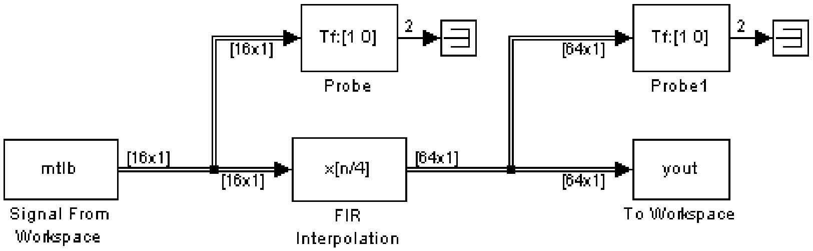
  The example below shows a single-channel input with a frame period of 1 second (**Sample time** = `1/64` and **Samples per frame** = `64` in the Signal From Workspace block) being interpolated by a factor of 4 to a frame period of 0.25 second. The input and output frame sizes are identical.



- `Maintain input frame rate`

  The block generates the output at the interpolated rate by using a proportionally larger frame *size* than the input. For interpolation by a factor of L, the output frame size is L times larger than the input frame size ($M_o = M_i*L$), but the input and output frame rates are equal.

  The example below shows a single-channel input of frame size 16 being interpolated by a factor of 4 to a frame size of 64. The block's input and output frame rates are identical.



Latency

Zero Latency. The FIR Interpolation block has *zero tasking latency* for all single-rate operations. The block is single-rate for the particular combinations of sampling mode and parameter settings shown in the table below.

Sampling Mode	Parameter Settings
Sample based	**Interpolation factor** parameter, L, is 1.
Frame based	**Interpolation factor** parameter, L, is 1, *or* **Framing** parameter is **Maintain input frame rate**.

Note that in sample-based mode, single-rate operation occurs only in the trivial case of factor-of-1 interpolation.

The block also has zero latency for sample-based multirate operations in the Simulink single-tasking mode. Zero tasking latency means that the block propagates the first filtered input (received at *t*=0) as the first input sample, followed by L-1 interpolated values, the second filtered input sample, and so on.

Nonzero Latency. The FIR Interpolation block is multirate for all settings other than those in the 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	None
Multitasking	L samples	L frames (M_i samples per frame)

If the block exhibits latency, the default initial condition is zero. Alternatively, you can enter a value in the **Output buffer initial conditions** text box. This value is divided by the **Interpolation factor** and output at the output port until the first filtered input sample is available.

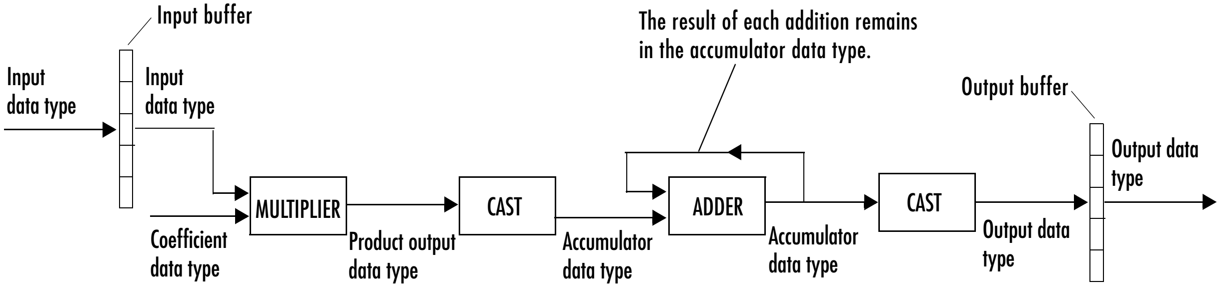
In sample-based cases, the scaled initial conditions appear at the start of each channel, followed immediately by the first filtered input sample, L-1 interpolated values, and so on.

In frame-based cases, with the default initial condition, the first M_iL output rows contain zeros, where M_i is the input frame size. The first filtered input sample (first filtered row of the input matrix) appears in the output as sample M_iL+1, followed by L-1 interpolated values, the second filtered input sample, and so on. See the example below for an illustration of this case.

See “Excess Algorithmic Delay (Tasking Latency)” on page 2-98 and the topic called The Simulation Parameters Dialog Box in the Simulink documentation for more information about block rates and the Simulink tasking modes.

Fixed-Point Data Types

The following diagram shows the data types used within the FIR Interpolation block for fixed-point signals:

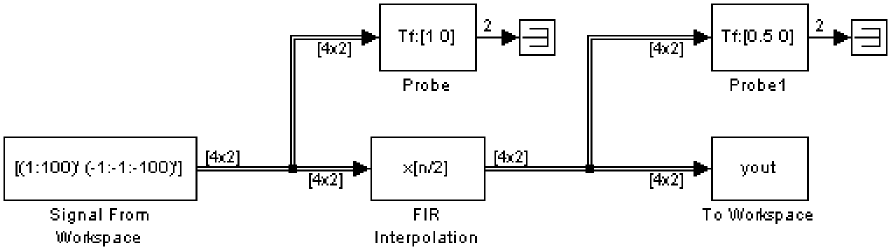


You can set the product output, accumulator, and output data types in the block mask as discussed in "Dialog Box" on page 7-353 below. The diagram shows that input data is stored in the input buffer in the same data type and scaling as the input. Filtered data is stored in the output buffer in the output data type and scaling that is set by you in the block mask. Any initial conditions are also stored in the output buffer in the output data type and scaling set by you in the block mask.

Examples

Example 1

Construct the frame-based model shown below.



Adjust the block parameters as follows:

- Configure the Signal From Workspace block to generate a two-channel signal with frame size of 4 and sample period of 0.25. This represents an output frame period of 1 (0.25∗4). The first channel should contain the positive ramp signal 1, 2, ..., 100, and the second channel should contain the negative ramp signal -1, -2, ..., -100.

  - **Signal** = `[(1:100)' (-1:-1:-100)']`
  - **Sample time** = `0.25`
  - **Samples per frame** = `4`

- Configure the FIR Interpolation block to interpolate the two-channel input by increasing the output frame rate by a factor of 2 relative to the input frame rate. Use a third-order filter (*m*=3) with normalized cutoff frequency, f_{n0}, of 0.25. (Note that f_{n0} and *m* satisfy $f_{n0} \leq 1/L$ and $m > L$.)
  - **FIR filter coefficients =** `fir1(3,0.25)`
  - **Interpolation factor** = `2`
  - **Framing** = `Maintain input frame size`

  The filter coefficient vector generated by `fir1(3,0.25)` is

  ```
  [0.0386 0.4614 0.4614 0.0386]
  ```

  or, equivalently,

  $$H(z) = B(z) = 0.0386 + 0.04614z^{-1} + 0.04614z^{-2} + 0.0386z^{-3}$$

- Configure the Probe blocks by clearing the **Probe width**, **Probe complex signal**, and **Probe signal dimensions** check boxes (if desired).

This model is multirate because there are at least two distinct sample rates, as shown by the two Probe blocks. To run this model in the Simulink multitasking mode, 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.

```
dsp_examples_yout =

            0          0
            0          0
            0          0
            0          0
            0          0
            0          0
```

```
      0         0
      0         0
 0.0386   -0.0386
 0.4614   -0.4614
 0.5386   -0.5386
 0.9614   -0.9614
 1.0386   -1.0386
```

Since we ran this frame-based multirate model in multitasking mode, the first eight (M_iL) output rows are zero. The first filtered input matrix row appears in the output as sample 9 (that is, sample M_iL+1). Every other row is an interpolated value.

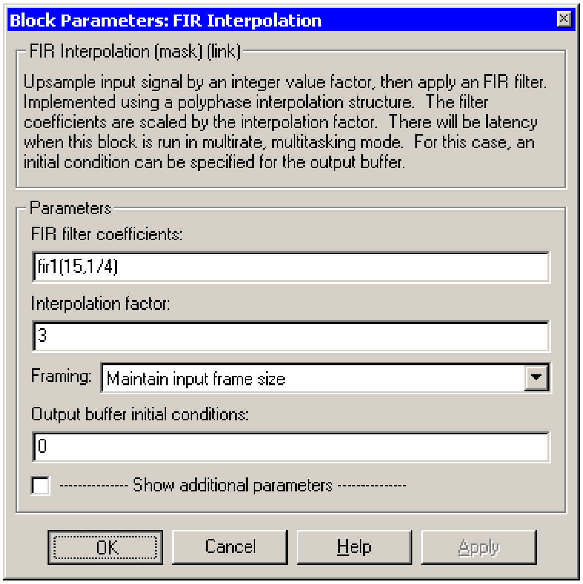
Example 2

The Polyphase FIR Interpolation demo (`polyphaseInterp_demo`) illustrates the underlying polyphase implementations of the FIR Interpolation block. Run the demo and view the results on the scope. The output of the FIR Interpolation block is the same as the output of the Polyphase Interpolation Filter block.

Example 3

The `dspintrp` demo provides another simple example, and the `dspmrf_menu` demo illustrates the use of the FIR Interpolation block in a number of multistage multirate filters.

Dialog Box



Block Parameters: FIR Interpolation

FIR Interpolation (mask) (link)

Upsample input signal by an integer value factor, then apply an FIR filter. Implemented using a polyphase interpolation structure. The filter coefficients are scaled by the interpolation factor. There will be latency when this block is run in multirate, multitasking mode. For this case, an initial condition can be specified for the output buffer.

Parameters

FIR filter coefficients: fir1(15,1/4)

Interpolation factor: 3

Framing: Maintain input frame size

Output buffer initial conditions: 0

□ ------------- Show additional parameters -------------

FIR filter coefficients

Specify the FIR filter coefficients, in descending powers of z.

Interpolation factor

Specify the integer factor, L, by which to increase the sample rate of the input sequence.

Framing

For frame-based operation, specify the method by which to implement the interpolation: increase the output frame rate, or increase the output frame size. This parameter may not be set to `Maintain input frame rate` for sample-based signals.

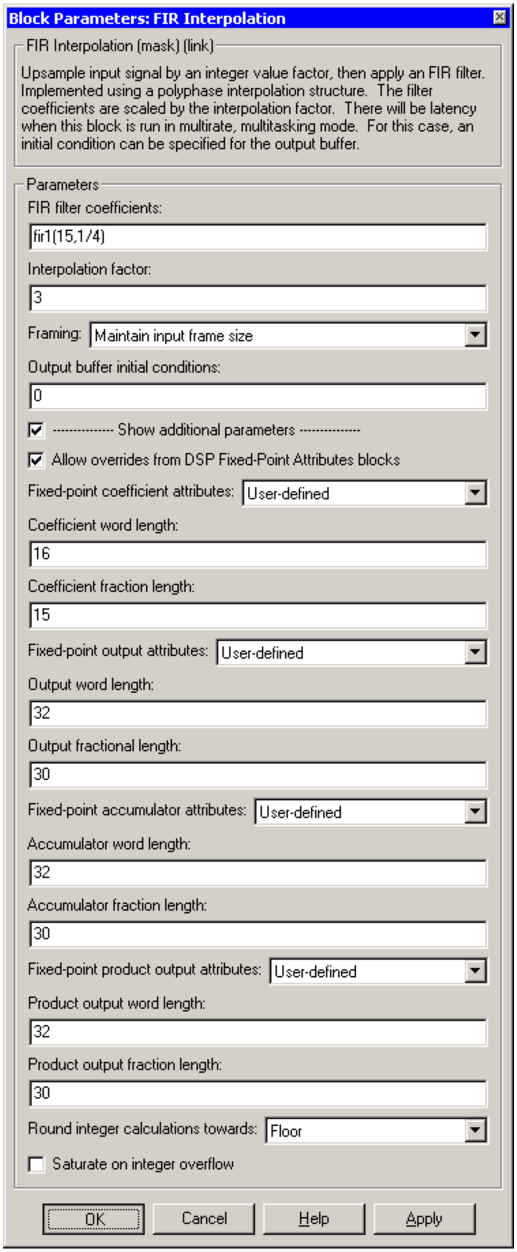
Output buffer initial conditions

If the block exhibits latency, enter a value in the **Output buffer initial conditions** text box to specify the value to output at the output port until the first filtered input sample is available. The default initial condition value is zero.

Output buffer initial conditions are stored in the output data type and scaling.

Show additional parameters

If selected, additional parameters specific to implementation of the block become visible as shown.



Block Parameters: FIR Interpolation

FIR Interpolation (mask) (link)

Upsample input signal by an integer value factor, then apply an FIR filter. Implemented using a polyphase interpolation structure. The filter coefficients are scaled by the interpolation factor. There will be latency when this block is run in multirate, multitasking mode. For this case, an initial condition can be specified for the output buffer.

Parameters

FIR filter coefficients: fir1(15,1/4)

Interpolation factor: 3

Framing: Maintain input frame size

Output buffer initial conditions: 0

☑ ------------- Show additional parameters -------------

☑ Allow overrides from DSP Fixed-Point Attributes blocks

Fixed-point coefficient attributes: User-defined

Coefficient word length: 16

Coefficient fraction length: 15

Fixed-point output attributes: User-defined

Output word length: 32

Output fractional length: 30

Fixed-point accumulator attributes: User-defined

Accumulator word length: 32

Accumulator fraction length: 30

Fixed-point product output attributes: User-defined

Product output word length: 32

Product output fraction length: 30

Round integer calculations towards: Floor

□ Saturate on integer overflow

Allow overrides from DSP Fixed-Point Attributes blocks

If you select this parameter, fixed-point data types for this block may be set by DSP Fixed-Point Attributes blocks in your model. If this parameter is unselected, the data types are always set by the parameters in the block mask.

Fixed-point coefficient attributes

Choose how you will specify the word length and fraction length of the filter coefficients. If you select `Same as input`, these characteristics will match those of the input to the block. If you select `User-defined`, the **Coefficient word length** and **Coefficient fraction length** parameters become visible.

The coefficients do not obey the **Round integer calculations toward** and the **Saturate on integer overflow** parameters; they are always saturated and rounded to `Nearest`.

Coefficient word length

Specify the word length, in bits, of the coefficients. This parameter is only visible if `User-defined` is specified for the **Fixed-point coefficient attributes** parameter.

Coefficient fraction length

Specify the fraction length, in bits, of the coefficients. This parameter is only visible if `User-defined` is specified for the **Fixed-point coefficient attributes** parameter.

Fixed-point output attributes

Choose how you will specify the output word length and fraction length. If you select `Same as input`, these characteristics will match those of the input to the block. If you select `User-defined`, the **Output word length** and **Output fraction length** parameters become visible.

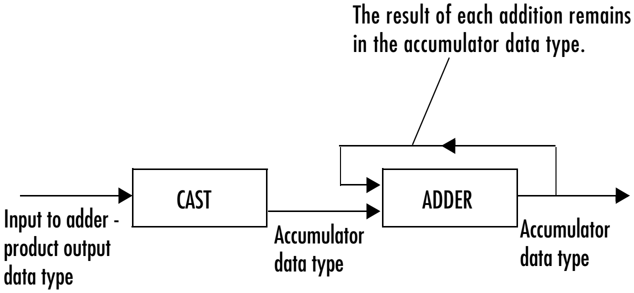
Output word length

Specify the word length, in bits, of the output. This parameter is only visible if `User-defined` is specified for the **Fixed-point output attributes** parameter.

Output fraction length

Specify the fraction length, in bits, of the output. This parameter is only visible if `User-defined` is specified for the **Fixed-point output attributes** parameter.

Fixed-point accumulator attributes


The result of each addition remains in the accumulator data type.
CAST
ADDER
Input to adder - product output data type
Accumulator data type
Accumulator data type

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

If you select `Same as output`, the accumulator word and fraction lengths are the same as those of the output of the block. If you select `User-defined`, the **Accumulator word length** and **Accumulator fraction length** parameters become visible.

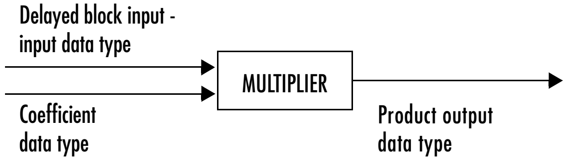
Accumulator word length

Specify the word length, in bits, of the accumulator. This parameter is only visible when `User-defined` is specified for the **Fixed-point accumulator attributes** parameter.

Accumulator fraction length

Specify the fraction length, in bits, of the accumulator. This parameter is only visible when `User-defined` is specified for the **Fixed-point accumulator attributes** parameter.

Fixed-point product output attributes



As depicted above, the output of the multiplier is placed into the product output data type and scaling. Use this parameter to specify how you would like to designate this product output word and fraction lengths.

If you select `Same as accumulator`, the product output word and fraction lengths are the same as those of the accumulator of the block. If you select `Same as output`, they are the same as those of the output of the block. If you select `User-defined`, the **Product output word length** and **Product output fraction length** parameters become visible.

Product output word length

Specify the word length, in bits, of the product output. This parameter is only visible when `User-defined` is specified for the **Fixed-point product output attributes** parameter.

Product output fraction length

Specify the fraction length, in bits, of the product output. This parameter is only visible when `User-defined` is specified for the **Fixed-point product output attributes** parameter.

Round integer calculations toward

Select the rounding mode for fixed-point operations. The filter coefficients do not obey this parameter; they always round to `Nearest`.

Saturate on integer overflow

If selected, overflows saturate. The filter coefficients do not obey this parameter; they are always saturated.

References

Fliege, N. J. *Multirate Digital Signal Processing: Multirate Systems, Filter Banks, Wavelets*. West Sussex, England: John Wiley & Sons, 1994.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point
- 8-, 16-, and 32-bit signed integers

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

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

FIR Rate Conversion

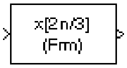
Purpose

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. See N.J. Fliege, *Multirate Digital Signal Processing: Multirate Systems, Filter Banks, Wavelets* for more information.

The **FIR filter coefficients** parameter specifies the numerator coefficients of the FIR filter transfer function H(*z*).

$$H(z) = B(z) = b_1 + b_2 z^{-1} + \ldots + b_m z^{-(m-1)}$$

The 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.

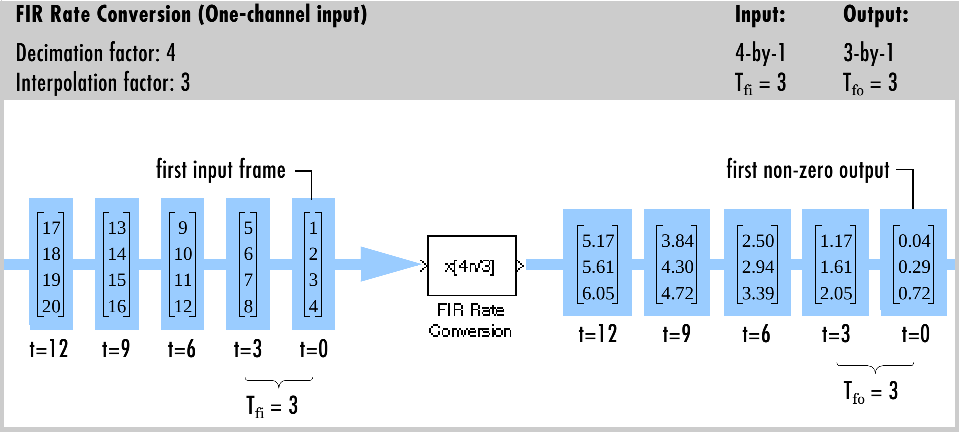
Frame-Based Operation

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

Example 1

The Rate Converter demo (`polyphaseUpFirDn_demo`) illustrates the underlying polyphase implementations of the FIR Rate Conversion block. Run the demo and view the results on the scope. The output of the FIR Rate Conversion block is the same as the output of the system comprised of the FIR Decimation block and FIR Interpolation block. The output of the FIR Rate Conversion block is also the same as the output of the Polyphase Filter block.

Example 2

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.

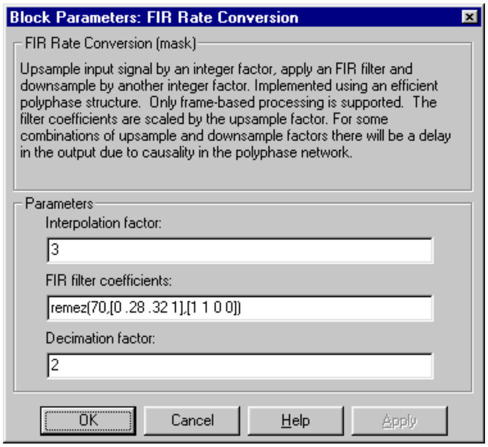
```
(Input port width)/(Output port width) must equal the
(Decimation factor)/(Interpolation factor).
```

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



Block Parameters: FIR Rate Conversion

FIR Rate Conversion (mask)

Upsample input signal by an integer factor, apply an FIR filter and downsample by another integer factor. Implemented using an efficient polyphase structure. Only frame-based processing is supported. The filter coefficients are scaled by the upsample factor. For some combinations of upsample and downsample factors there will be a delay in the output due to causality in the polyphase network.

Parameters

Interpolation factor: 3

FIR filter coefficients: remez(70,[0 .28 .32 1],[1 1 0 0])

Decimation factor: 2

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 your data types to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-2.

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 2-19
- “Multirate Filters” on page 3-61

Flip

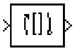
Purpose

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
```

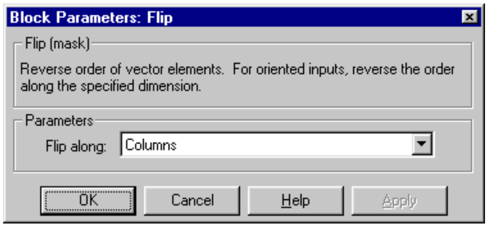
For convenience, length-M 1-D vector inputs are treated as M-by-1 column vectors for vertical flipping.

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



Block Parameters: Flip

Flip (mask)

Reverse order of vector elements. For oriented inputs, reverse the order along the specified dimension.

Parameters

Flip along: Columns

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 your data types to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-2.

See Also

Selector	Simulink
Transpose	DSP Blockset
Variable Selector	DSP Blockset
`flipud`	MATLAB
`fliplr`	MATLAB

Forward Substitution

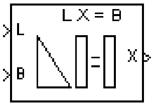
Purpose

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

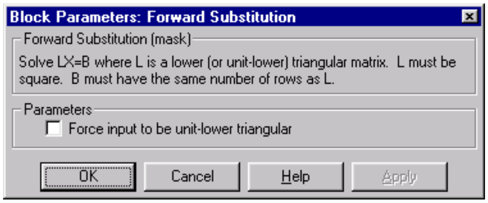


The Forward Substitution block solves the linear system L*X*=B by simple forward substitution of variables, where L is the lower triangular M-by-M matrix input to the `L` port, and B is the M-by-N matrix input to the `B` port. The output is the solution of the equations, the M-by-N matrix *X,* and is 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 1's. This is useful when matrix L is the result of another operation, such as an LDL decomposition, that uses the diagonal elements to represent the D matrix.

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

Dialog Box



Forward Substitution (mask)

Solve LX=B where L is a lower (or unit-lower) triangular matrix. L must be square. B must have the same number of rows as L.

Parameters

□ Force input to be unit-lower triangular

Force input to be unit-lower triangular

Replaces the elements on the diagonal of L with 1's when selected. Tunable.

Supported Data Types

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

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

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 5-6 for related information.

Frame Status Conversion

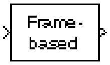
Purpose

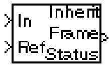
Specify the frame status of the output as sample based or frame based

Library

Signal Management / Signal Attributes

Description



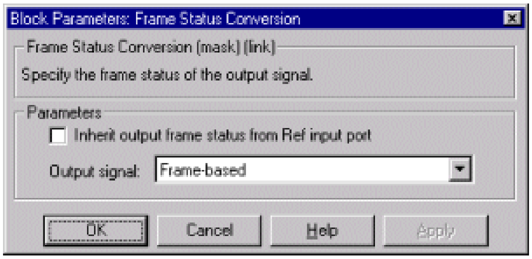


The Frame Status Conversion block passes the input through to the output, and sets the output frame status to the **Output signal** parameter, which can be either `Frame-based` or `Sample-based`. The output frame status can also be inherited from the signal at the `Ref` (reference) input port, which 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 (that is, 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



Block Parameters: Frame Status Conversion

Frame Status Conversion (mask) (link)

Specify the frame status of the output signal.

Parameters

☐ Inherit output frame status from Ref input port

Output signal: Frame-based

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 your data types to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-2.

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

From Wave Device

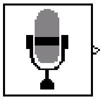
Purpose

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 pulse code modulation (PCM) format, and should typically be sampled at one of the standard Windows audio device rates: 8000, 11025, 22050, or 44100 Hz. You can select one of these rates from the **Sample rate** parameter. To specify a different rate, select the **User-defined** option and enter a value in the **User-defined sample rate** parameter.

The **Sample Width (bits)** parameter specifies the number of bits used to represent the signal samples read by the audio device. The following settings are available:

- `8` — allocates 8 bits to each sample, allowing a resolution of 256 levels
- `16` — allocates 16 bits to each sample, allowing a resolution of 65536 levels

- 24 — allocates 24 bits to each sample, allowing a resolution of 16777216 levels (only for use with 24-bit audio devices)

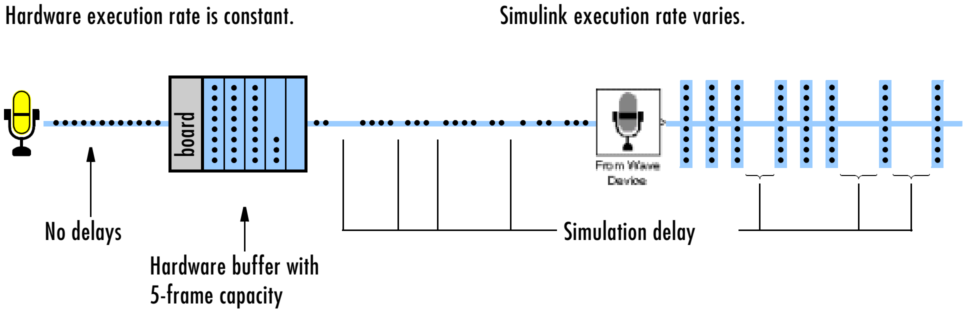
Higher sample width settings require more memory but yield better fidelity. The output from the block is independent of the **Sample width (bits)** setting. The output data type is determined by the **Data type** parameter setting.

Buffering

Since the audio device accepts real-time audio input, Simulink must read a continuous stream of data from the device throughout the simulation. Delays in reading data from the audio hardware can result in hardware errors or distortion of the signal. This means that the From Wave Device block must read data from the audio hardware as quickly as the hardware itself acquires the signal. However, the block often *cannot* match the throughput rate of the audio hardware, especially when the simulation is running from within Simulink rather than as generated code. (Simulink operations are generally slower than comparable hardware operations, and execution speed routinely varies during the simulation as the host operating system services other processes.) The block must therefore rely on a buffering strategy to ensure that signal data can be read on schedule without losing samples.

At the start of the simulation, the audio device begins writing the input data to a (hardware) buffer with a capacity of T_b seconds. The From Wave Device block immediately begins pulling the earliest samples off the buffer (first in, first out) and collecting them in length-M frames for output. As the audio device continues to append inputs to the bottom of the buffer, the From Wave Device block continues to pull inputs off the top of the buffer at the best possible rate.

The following figure shows an audio signal being acquired and output with a frame size of 8 samples. The buffer of the sound board is approaching its five-frame capacity at the instant shown, which means that the hardware is adding samples to the buffer more rapidly than the block is pulling them off. (If the signal sample rate was 8 kHz, this small buffer could hold approximately 0.005 second of data.


Hardware execution rate is constant.
Simulink execution rate varies.
board
From Wave Device
No delays
Hardware buffer with 5-frame capacity
Simulation delay

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.

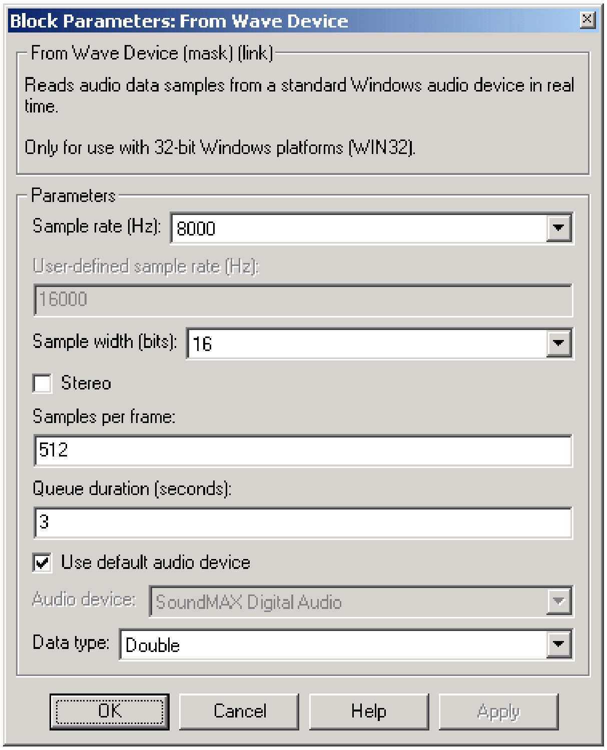
- *Increase the simulation throughput rate*

  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 2-92, and “Improving Simulation Performance and Accuracy” in the Simulink documentation, for other ideas on improving simulation performance.

Dialog Box



Block Parameters: From Wave Device

From Wave Device (mask) (link)

Reads audio data samples from a standard Windows audio device in real time.

Only for use with 32-bit Windows platforms (WIN32).

Parameters

Sample rate (Hz): 8000

User-defined sample rate (Hz): 16000

Sample width (bits): 16

□ Stereo

Samples per frame: 512

Queue duration (seconds): 3

☑ Use default audio device

Audio device: SoundMAX Digital Audio

Data type: Double

Sample rate (Hz)

The sample rate of the audio data to be acquired. Select one of the standard Windows rates or the `User-defined` option.

User-defined sample rate (Hz)

The (nonstandard) sample rate of the audio data to be acquired.

Sample width (bits)

The number of bits used to represent each signal sample.

Stereo

Specifies stereo (two-channel) inputs when selected, mono (one-channel) inputs when cleared. Stereo output is M-by-2; mono output is M-by-1.

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

See Also

From Wave File	DSP Blockset
To Wave Device	DSP Blockset
`audiorecorder`	MATLAB
`audiodevinfo`	MATLAB

See "Importing WAV Files" on page 2-78 for related information.

From Wave File

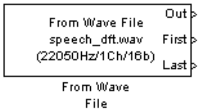
Purpose

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 pulse code modulation (PCM) format.

```
  y = wavread('filename')    % Equivalent MATLAB code
```

The block supports 8-, 16-, 24-, and 32-bit Microsoft Wave (.wav) files.

The **File name** parameter can specify an absolute or relative path to the file. 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.

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 output frame** parameter. For M=1, the output is sample based; otherwise, the output is frame based.

The output frame period, T_{fo}, is

$$T_{fo} = \frac{M}{F_s},$$

where F_s is the data sample rate in Hz.

To reduce the required number of file accesses, the block acquires L consecutive samples from the file during each access, where L is specified by the **Minimum number of samples for each read from file** parameter (L ≥ 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.

Select the **Loop** check box if you want to play the file more than once. Then, enter the number of times to play the file. The number you enter must be a positive integer or inf.

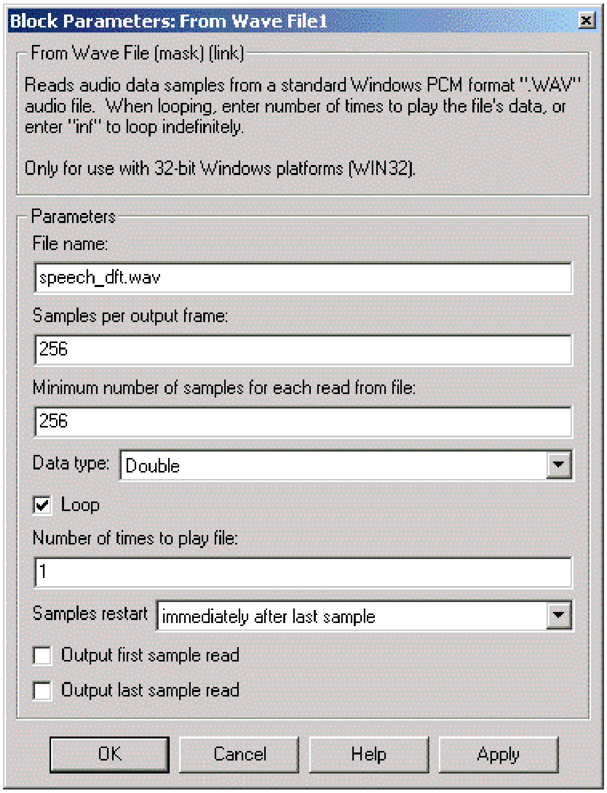
The **Samples restart** parameter determines whether the samples from the audio file repeat immediately or repeat at the beginning of the next frame output from the output port. If you select `immediately after last sample`, the samples repeat immediately. If you select `at beginning of next frame`, the frame containing the last sample value from the audio file is zero padded until the frame is filled. The block then places the first sample of the audio file in the first position of the next output frame.

Use the **Output first sample read** parameter to determine when the first sample of the audio file is contained within an output frame. If you select this check box, a Boolean output port labeled First appears on the From Wave File block. The output from the First port is 1 when the frame output from the output port contains the first sample of the audio file. Otherwise, the output from the First port is 0.

Use the **Output last sample read** parameter to determine when the last sample of the audio file is contained within an output frame. If you select this check box, a Boolean output port labeled Last appears on the From Wave File block. The output from the Last port is 1 when the frame output from the output port contains the last sample of the audio file. Otherwise, the output from the Last port is 0.

The block icon shows the name, sample rate (in Hz), number of channels (1 or 2), and sample width (in bits) of the data in the specified audio file. All sample rates are supported; the sample width must be either 8, 16, 24, or 32 bits.

Dialog Box



Block Parameters: From Wave File1

From Wave File (mask) (link)

Reads audio data samples from a standard Windows PCM format ".WAV" audio file. When looping, enter number of times to play the file's data, or enter "inf" to loop indefinitely.

Only for use with 32-bit Windows platforms (WIN32).

Parameters

File name: speech_dft.wav

Samples per output frame: 256

Minimum number of samples for each read from file: 256

Data type: Double

☑ Loop

Number of times to play file: 1

Samples restart: immediately after last sample

□ Output first sample read

□ Output last sample read

File name

Enter the path and name of the file to read. Paths can be relative or absolute.

Samples per output frame

Enter the number of samples in each output frame, M.

Minimum number of samples for each read from file

Enter the number of consecutive samples to acquire from the file with each file access, L.

Data type

Select the output data type: `Double`, `Single`, `Uint8`, or `Int16`. The data type setting determines the output's amplitude range, as shown in the table above.

Loop

Select this check box if you want to play the file more than once.

Number of times to play file

Enter the number of times you want to play the file.

Samples restart

Select `immediately after last sample` to repeat the audio file immediately. Select `at beginning of next frame` to place the first sample of the audio file in the first position of the next output frame.

Output first sample read

Use this check box to determine whether the frame output from the output port contains the first sample of the audio file.

Output last sample read

Use this check box to determine whether the frame output from the output port contains the last sample of the audio file.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- 16-bit signed integer
- 8-bit unsigned integer

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

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 2-78 for related information.

Histogram

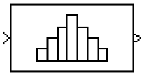
Purpose

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

$$B_m + \left(k + \frac{1}{2}\right)\Delta \qquad\qquad k = 0, 1, 2, \ldots, n-1$$

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.

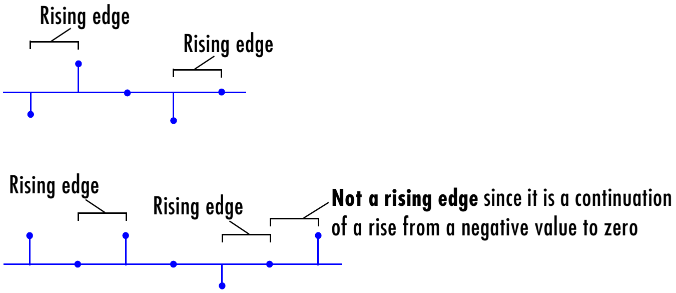
As in basic operation, length-M 1-D vector inputs and *sample-based* length-M row vector inputs are both treated as M-by-1 column vectors.

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.

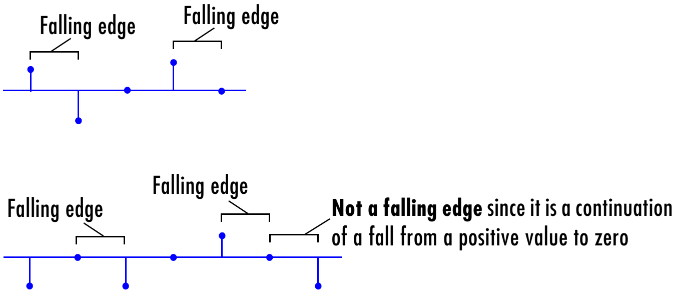
Resetting the Running Histogram. The block resets the running histogram whenever a reset event is detected at the optional Rst port. The reset signal rate must be a positive integer multiple of the rate of the data signal input.

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)


Rising edge
Rising edge
Rising edge
Rising edge
Not a rising edge since it is a continuation
of a rise from a negative value to zero

- `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)


Falling edge
Falling edge
Falling edge
Falling edge
Not a falling edge since it is a continuation
of a fall from a positive value to zero

- `Either edge` — Triggers a reset operation when the `Rst` input is a `Rising edge` or `Falling edge` (as described above).
- `Non-zero sample` — Triggers a reset operation at each sample time that the `Rst` input is not zero.

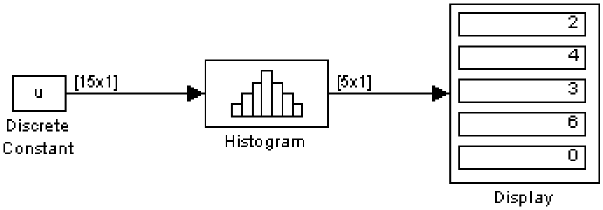
Note When running simulations in the Simulink `MultiTasking` mode, sample-based reset signals have a one-sample latency, and frame-based reset signals have one frame of latency. Thus, there is a one-sample or one-frame delay between the time the block detects a reset event, and when it applies the reset. For more information on latency and the Simulink tasking modes, see

"Excess Algorithmic Delay (Tasking Latency)" on page 2-98 and the topic called The Simulation Parameters Dialog Box in the Simulink documentation.

Examples

The model below illustrates the Histogram block's basic operation for a single-channel input, u, where

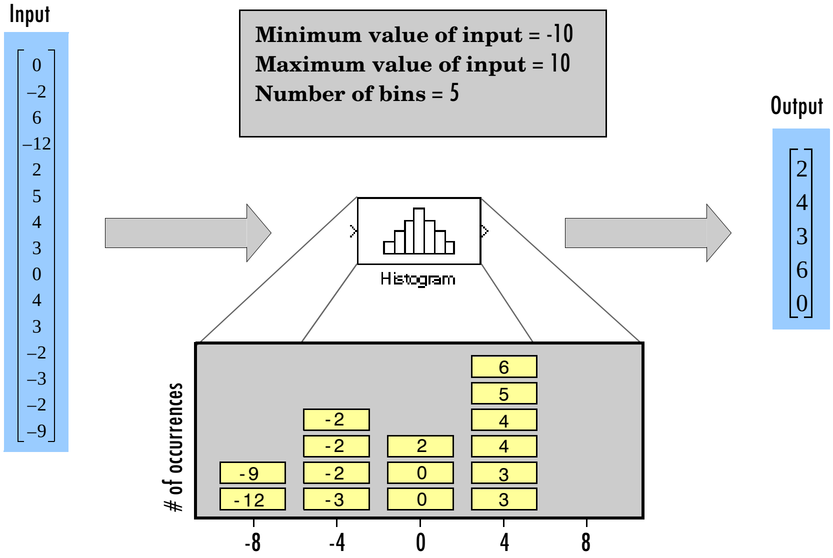
```
u = [0 -2 6 -12 2 5 4 3 0 4 3 -2 -3 -2 -9]'
```



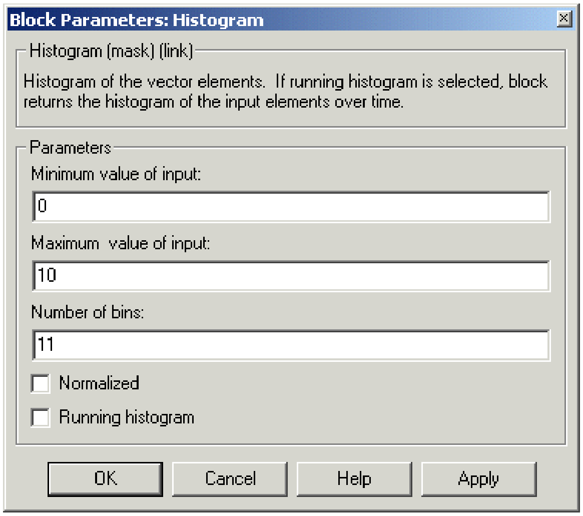
The parameter settings for the Histogram block are

- **Minimum value of input** = -10
- **Maximum value of input** = 10
- **Number of bins** = 5
- **Normalized** = Clear this check box
- **Running histogram** = Clear this check box

The resulting bin width is 4, as shown below.



Dialog Box



Block Parameters: Histogram

Histogram (mask) (link)

Histogram of the vector elements. If running histogram is selected, block returns the histogram of the input elements over time.

Parameters

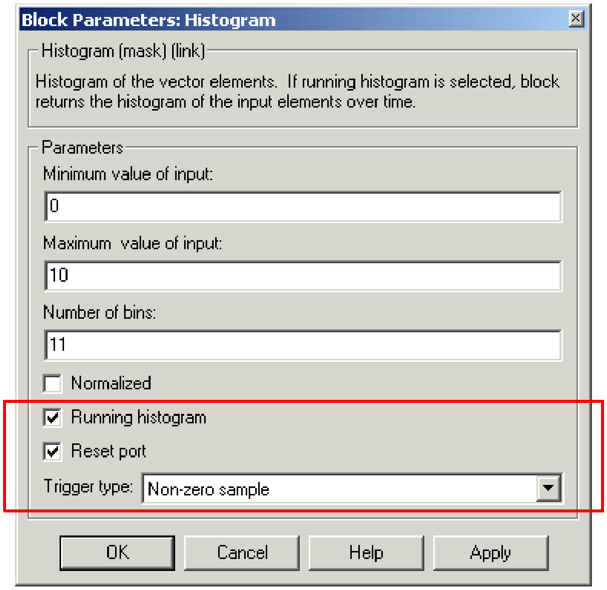
Minimum value of input: 0

Maximum value of input: 10

Number of bins: 11

□ Normalized

□ Running histogram



Block Parameters: Histogram

Histogram (mask) (link)

Histogram of the vector elements. If running histogram is selected, block returns the histogram of the input elements over time.

Parameters

Minimum value of input: 0

Maximum value of input: 10

Number of bins: 11

□ Normalized

☑ Running histogram

☑ Reset port

Trigger type: Non-zero sample

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 7-380 and "Running Operation" on page 7-381.

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 7-381.

Trigger type

The type of event that resets the running histogram. For more information, see "Resetting the Running Histogram" on page 7-381. 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 your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-2.

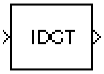
See Also

Sort	DSP Blockset
`hist`	MATLAB

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 *l*th column contains the length-M IDCT of the corresponding input column.

$$y(m,l) = \sum_{k=1}^{M} w(k)u(k,l)\cos\frac{\pi(2m-1)(k-1)}{2M}, \qquad m = 1, \ldots, M$$

where

$$w(k) = \begin{cases} \dfrac{1}{\sqrt{M}}, & k = 1 \\ \sqrt{\dfrac{2}{M}}, & 2 \le k \le M \end{cases}$$

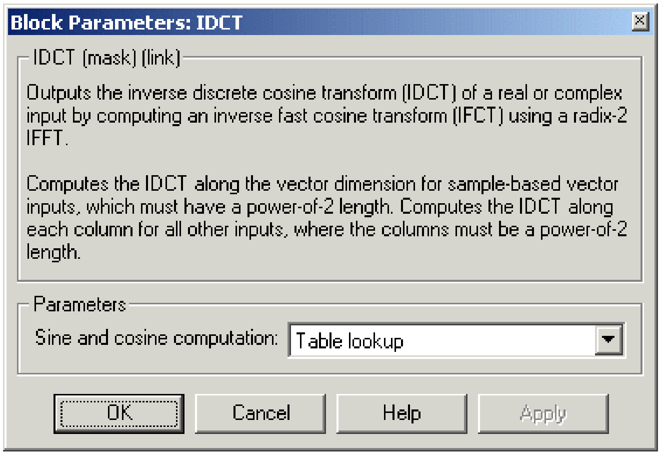
The output is always frame based, and the output sample rate and data type (real/complex) are the same as those of the input.

For convenience, length-M 1-D vector inputs and *sample-based* length-M row vector inputs are processed as single channels (that is, as M-by-1 column vectors), and the output has the same dimension as the input.

The **Sine and cosine computation** parameter determines how the block computes the necessary sine and cosine values 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



Block Parameters: IDCT

IDCT (mask) (link)

Outputs the inverse discrete cosine transform (IDCT) of a real or complex input by computing an inverse fast cosine transform (IFCT) using a radix-2 IFFT.

Computes the IDCT along the vector dimension for sample-based vector inputs, which must have a power-of-2 length. Computes the IDCT along each column for all other inputs, where the columns must be a power-of-2 length.

Parameters

Sine and cosine computation: Table lookup

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 your data types to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-2.

See Also

DCT	DSP Blockset
IFFT	DSP Blockset
`idct`	Signal Processing Toolbox

Identity Matrix

Purpose

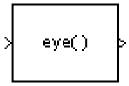
Generate a matrix with ones on the main diagonal and zeros elsewhere

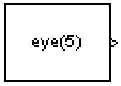
Library

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
```

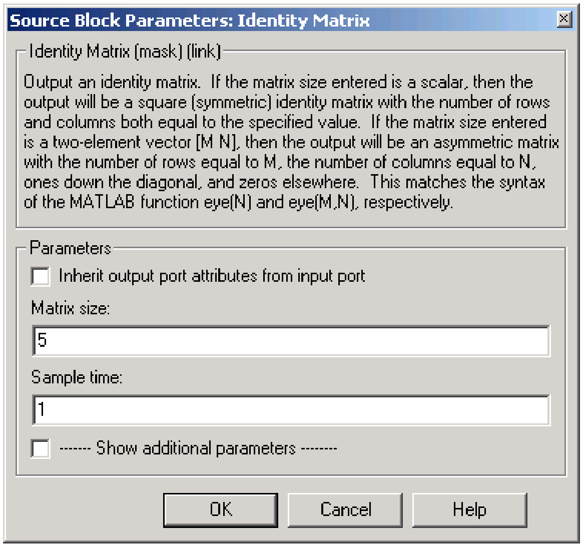
When the **Inherit output port attributes from input port** check box is *not* selected, the input port is disabled, and the dimensions of the output matrix are determined by the **Matrix size** parameter. A scalar value, `M`, specifies an M-by-M identity matrix, while a two-element vector, `[M N]`, specifies an M-by-N unit-diagonal matrix. The output is sample based, and has the sample period specified by the **Sample time** parameter.

Examples

Set **Matrix size** to `[3 6]` to generate the 3-by-6 unit-diagonal matrix below.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Dialog Box



Source Block Parameters: Identity Matrix

Identity Matrix (mask) (link)

Output an identity matrix. If the matrix size entered is a scalar, then the output will be a square (symmetric) identity matrix with the number of rows and columns both equal to the specified value. If the matrix size entered is a two-element vector [M N], then the output will be an asymmetric matrix with the number of rows equal to M, the number of columns equal to N, ones down the diagonal, and zeros elsewhere. This matches the syntax of the MATLAB function eye(N) and eye(M,N), respectively.

Parameters

□ Inherit output port attributes from input port

Matrix size:

5

Sample time:

1

□ ------- Show additional parameters --------

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

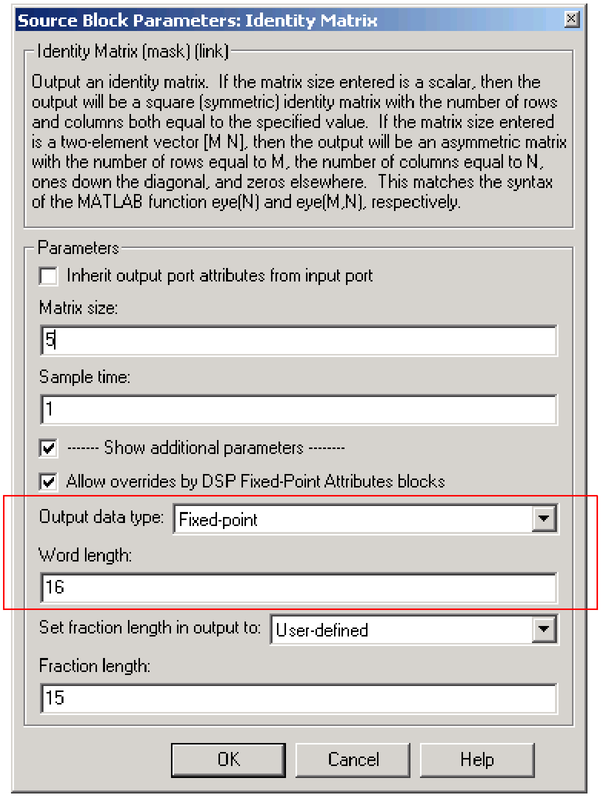
The number of rows and columns in the output matrix: a scalar `M` for a square M-by-M output, or a vector `[M N]` for an M-by-N output. This parameter is disabled when **Inherit input port attributes from input port** is selected.

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.



Source Block Parameters: Identity Matrix

Identity Matrix (mask) (link)

Output an identity matrix. If the matrix size entered is a scalar, then the output will be a square (symmetric) identity matrix with the number of rows and columns both equal to the specified value. If the matrix size entered is a two-element vector [M N], then the output will be an asymmetric matrix with the number of rows equal to M, the number of columns equal to N, ones down the diagonal, and zeros elsewhere. This matches the syntax of the MATLAB function eye(N) and eye(M,N), respectively.

Parameters

□ Inherit output port attributes from input port

Matrix size: 5

Sample time: 1

☑ ------- Show additional parameters -------

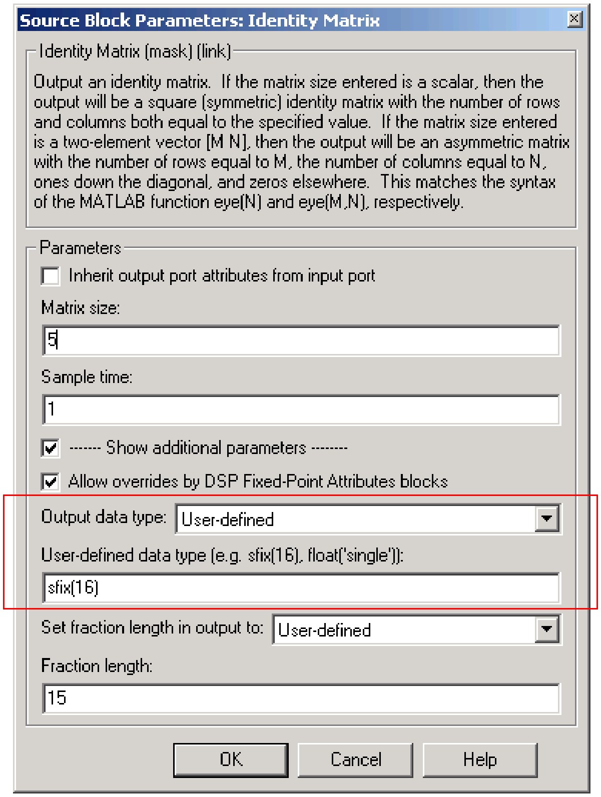
☑ Allow overrides by DSP Fixed-Point Attributes blocks

Output data type: Fixed-point

Word length: 16

Set fraction length in output to: User-defined

Fraction length: 15



Source Block Parameters: Identity Matrix

Identity Matrix (mask) (link)

Output an identity matrix. If the matrix size entered is a scalar, then the output will be a square (symmetric) identity matrix with the number of rows and columns both equal to the specified value. If the matrix size entered is a two-element vector [M N], then the output will be an asymmetric matrix with the number of rows equal to M, the number of columns equal to N, ones down the diagonal, and zeros elsewhere. This matches the syntax of the MATLAB function eye(N) and eye(M,N), respectively.

Parameters

□ Inherit output port attributes from input port

Matrix size: 5

Sample time: 1

☑ ------- Show additional parameters -------

☑ Allow overrides by DSP Fixed-Point Attributes blocks

Output data type: User-defined

User-defined data type (e.g. sfix(16), float('single')): sfix(16)

Set fraction length in output to: User-defined

Fraction length: 15

Allow overrides from DSP Fixed-Point Attributes blocks

If you select this parameter, and if the **Output data type parameter** is set to `Fixed-point`, fixed-point data types for this block may be set by DSP Fixed-Point Attributes blocks in your model. If this parameter is unselected, the data types are always set by the parameters in the block mask.

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 fraction length in output to,** and **Fraction length** parameters.
- Choose `User-defined` to specify the output data type and scaling in the **User-defined data type**, **Set fraction length in output to,** and **Fraction length** parameters.
- Choose `Inherit via back propagation` to set the output data type and scaling to match the following block

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 fraction length in output to

Specify the scaling of the fixed-point output by either of the following two methods:

- Choose `Best precision` to have the output scaling automatically set such that the output signal has the best possible precision.
- Choose `User-defined` to specify the output scaling in the **Fraction length** parameter.

This parameter is only visible if `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.

Fraction length

For fixed-point output data types, specify the number of fractional bits, or bits to the right of the binary point. This parameter is only visible if `Fixed-point` or `User-defined` is selected for the **Output data type** parameter, and if `User-defined` is selected for the **Set fraction length in output to** parameter.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point
- Custom data types
- Boolean
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

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

See Also

Constant Diagonal Matrix	DSP Blockset
DSP Constant	DSP Blockset
`eye`	MATLAB

Also “Creating Signals Using Constant Blocks” on page 2-32 for how to use this and other blocks to generate constant signals.

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" on page 7-271 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" on page 7-271.

Examples

See the examples in the Dyadic Synthesis Filter Bank block reference.

See Also

Dyadic Synthesis Filter Bank	DSP Blockset

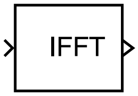
Purpose

Compute the IFFT of the input

Library

Transforms

Description



The IFFT block computes the inverse fast Fourier transform (IFFT) of each channel of an M-by-N or length-M input, `u`, where M must be a power of two. To work with other input sizes, use the Zero Pad block to pad or truncate the length-M dimension to a power-of-two length.

The output of the IFFT block is equivalent to the MATLAB `ifft` function:

```
y = ifft(u,M)      % Equivalent MATLAB code
```

The kth entry of the lth output channel, $y(k, l)$, is equal to the kth point of the M-point inverse discrete Fourier transform (IDFT) of the lth input channel:

$$y(k,l) = \frac{1}{M}\sum_{m=1}^{M} u(m,l)e^{j2\pi(m-1)(k-1)/M} \qquad k = 1, \ldots, M$$

This block supports real and complex floating-point and fixed-point inputs.

Sections of This Reference Page

For information on block output characteristics and how to configure the block computation methods, see other sections of this reference page:

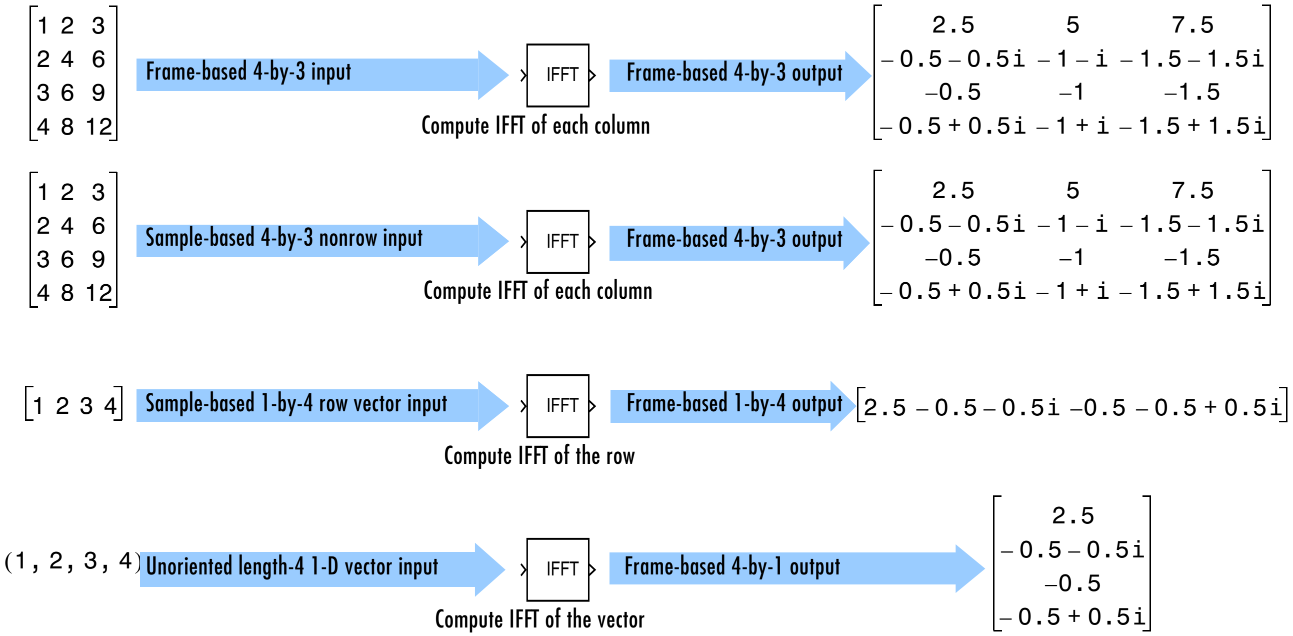
- “Input and Output Characteristics” on page 7-397
- “Selecting the Twiddle Factor Computation Method” on page 7-399
- “Optimizing the Table of Trigonometric Values” on page 7-399
- “Input Order” on page 7-400
- “Conjugate Symmetric Input” on page 7-401
- “Scaled Output” on page 7-401
- “Algorithms Used for IFFT Computation” on page 7-401
- “Examples” on page 7-404
- “Dialog Box” on page 7-404
- “Supported Data Types” on page 7-409
- “See Also” on page 7-409

Input and Output Characteristics

The following table describes valid inputs to the IFFT block, their corresponding outputs, and the dimension along which the block computes the IDFT.

Valid Block Inputs • Real- or complex-valued • M must be a power of two • In linear or bit-reversed order	**Dimension Along Which Block Computes IDFT**	**Corresponding Block Output Characteristics** Output port rate = input port rate
Frame-based M-by-N matrix	Column	The following output characteristics apply to all valid block inputs: • Frame based • Complex valued, unless you select the **Input is conjugate symmetric** check box when your input is conjugate symmetric, in which case the output is real valued. The **Input is conjugate symmetric** check box cannot be used for fixed-point signals. • 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 **Skip normalization by transform length, N** check box is selected, rather than computing the IDFT, the block computes a scaled version of the IDFT. This scaled version of the IDFT does not include the multiplication factor of 1/M.
Sample-based M-by-N matrix, $M \neq 1$	Column	
Sample-based 1-by-M *row vector*	Row	
1-D length-M vector	Vector	

The following diagram shows the effects of the input size, dimension, and frame status on the output of the IFFT block (see also `ifft_ins_outs.mdl`).



[1 2 3; 2 4 6; 3 6 9; 4 8 12]
Frame-based 4-by-3 input
IFFT
Frame-based 4-by-3 output
Compute IFFT of each column
[2.5 5 7.5; −0.5 − 0.5i −1 − i −1.5 − 1.5i; −0.5 −1 −1.5; −0.5 + 0.5i −1 + i −1.5 + 1.5i]
[1 2 3; 2 4 6; 3 6 9; 4 8 12]
Sample-based 4-by-3 nonrow input
IFFT
Frame-based 4-by-3 output
Compute IFFT of each column
[2.5 5 7.5; −0.5 − 0.5i −1 − i −1.5 − 1.5i; −0.5 −1 −1.5; −0.5 + 0.5i −1 + i −1.5 + 1.5i]
[1 2 3 4]
Sample-based 1-by-4 row vector input
IFFT
Frame-based 1-by-4 output
Compute IFFT of the row
[2.5 −0.5 − 0.5i −0.5 −0.5 + 0.5i]
(1, 2, 3, 4)
Unoriented length-4 1-D vector input
IFFT
Frame-based 4-by-1 output
Compute IFFT of the vector
[2.5; −0.5 − 0.5i; −0.5; −0.5 + 0.5i]

Selecting the Twiddle Factor Computation Method

The **Twiddle factor computation** parameter determines how the block computes the necessary sine and cosine terms to calculate the term $e^{j2\pi(m-1)(k-1)/M}$, shown in the first equation of this block reference page. This parameter has two settings, each with its advantages and disadvantages, as described in the following table. Note that only `Table lookup` mode is supported for fixed-point signals.

Twiddle Factor Computation Parameter Setting	Sine and Cosine Computation Method	Effect on Block Performance
`Table lookup`	The block computes and stores the trigonometric values before the simulation starts, and retrieves them during the simulation. When you generate code from the block, the processor running the generated code stores the trigonometric values computed by the block, and retrieves the values during code execution.	The block usually runs much more quickly, but requires extra memory for storing the precomputed trigonometric values. You can optimize the table for memory consumption or speed, as described in “Optimizing the Table of Trigonometric Values” 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 also set the **Optimize table for** parameter. This parameter optimizes the table of trigonometric values for speed or memory by varying the number of table entries as summarized in the following table.

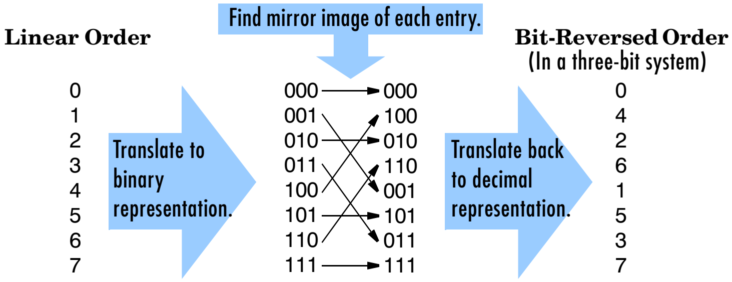
Optimize Table for Parameter Setting	Number of Table Entries for N-Point FFT	Memory Required for Single-Precision 512-Point FFT
`Speed`	$3N/4$ — floating point N — fixed point	1536 bytes
`Memory`	$N/4 + 1$ — floating point Not supported for fixed point	516 bytes

Input Order

You must select the **Input is in bit-reversed order** check box to designate whether the ordering of the column elements of the input is linear or bit-reversed.

Two numbers are bit-reversed values of each other when the binary representation of one is the mirror image of the binary representation of the other. For example, in a three-bit system, one and four are bit-reversed values of each other, since the three-bit binary representation of one, 001, is the mirror image of the three-bit binary representation of four, 100.

In the diagram below, the 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 finally translating the sequence back to its decimal representation. The resulting sequence is the original linearly ordered sequence in bit-reversed order.



Set the **Input is in bit-reversed order** parameter as follows to indicate the ordering of the input's column elements.

Parameter Setting	Ordering of Input Channel Elements
☐ Input is in bit-reversed order	Linear order
☑ Input is in bit-reversed order	Bit-reversed order

Conjugate Symmetric Input

When the block input is both floating point and conjugate symmetric and you want real-valued outputs, select the **Input is conjugate symmetric** check box. This 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.

If the IFFT block input is conjugate symmetric but you do not select the **Input is conjugate symmetric** check box, 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.

Note that the **Input is conjugate symmetric** parameter cannot be used for fixed-point signals.

Scaled Output

You can choose to output a scaled version of the input's IDFT, $M \cdot y(k, l)$, by setting the **Skip normalization by transform length, N** parameter.

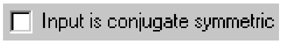
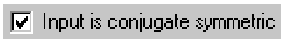
$$M \cdot y(k, l) = \sum_{m=1}^{M} u(m, l)e^{j2\pi(m-1)(k-1)/M} \quad k = 1, \ldots, M$$

Algorithms Used for IFFT Computation

Depending on whether the block input is floating point or fixed point, real or complex valued, and conjugate symmetric, the block uses one or more of the following algorithms as summarized in the following tables:

- Radix-2 decimation-in-time (DIT) algorithm
- Half-length algorithm
- Double-signal algorithm

For floating-point signals:

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

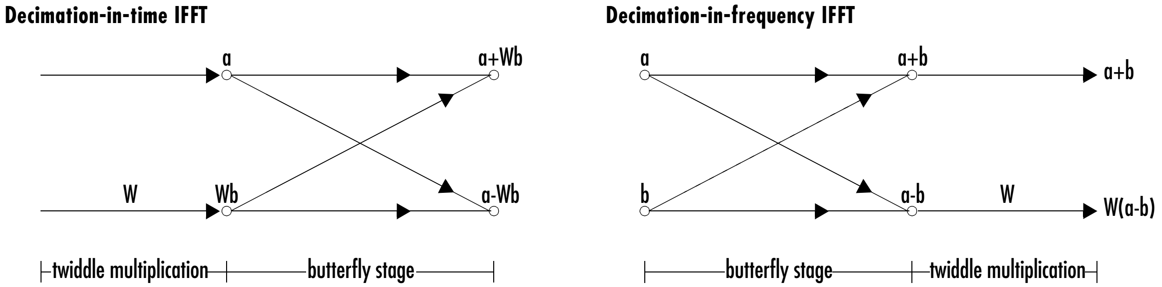
For fixed-point signals:

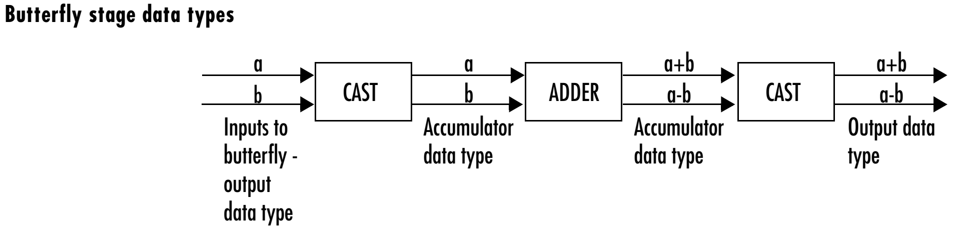
Input Complexity	Other Parameter Settings	Algorithms Used for FFT Computation
Real or complex	Not applicable	Radix-2 DIT

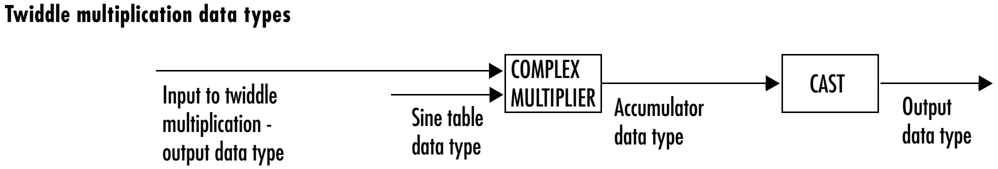
Fixed-Point Data Types

The diagrams below show the data types used within the IFFT block for fixed-point signals. You can set the sine table, accumulator, product output, and output data types displayed in the diagrams in the IFFT block mask as discussed in “Dialog Box” on page 7-404.

Inputs to the IFFT block are first cast to the output data type and stored in the output buffer. Each butterfly stage then processes signals in the accumulator data type, with the final output of the butterfly being cast back into the output data type. A twiddle factor is multiplied in before each butterfly stage in a decimation-in-time IFFT, and after each butterfly stage in a decimation-in-frequency IFFT.





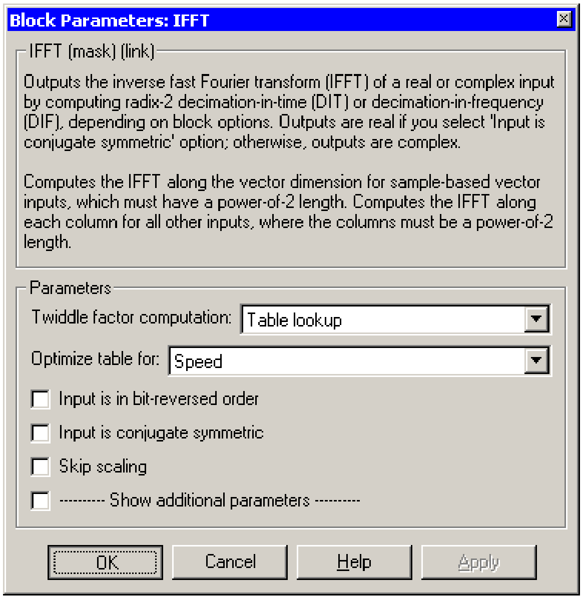

Twiddle multiplication data types
Input to twiddle multiplication - output data type
Sine table data type
COMPLEX MULTIPLIER
Accumulator data type
CAST
Output data type

The output of the multiplier is in the accumulator data type since both of the inputs to the multiplier are complex. For details on the complex multiplication performed, refer to “Multiplication Data Types” on page 6-15.

Examples

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, “The Use of Bit-Reversed Outputs” on page 7-309

Dialog Box



Block Parameters: IFFT

IFFT (mask) (link)

Outputs the inverse fast Fourier transform (IFFT) of a real or complex input by computing radix-2 decimation-in-time (DIT) or decimation-in-frequency (DIF), depending on block options. Outputs are real if you select 'Input is conjugate symmetric' option; otherwise, outputs are complex.

Computes the IFFT along the vector dimension for sample-based vector inputs, which must have a power-of-2 length. Computes the IFFT along each column for all other inputs, where the columns must be a power-of-2 length.

Parameters

Twiddle factor computation: Table lookup

Optimize table for: Speed

□ Input is in bit-reversed order

□ Input is conjugate symmetric

□ Skip scaling

□ --------- Show additional parameters ---------

Twiddle factor computation

Specify the computation method of the term $e^{j2\pi(m-1)(k-1)/M}$, shown in the first equation of this block reference page. In `Table lookup` mode, the block computes and stores the sine and cosine values before the simulation starts. In `Trigonometric fcn` mode, the block computes the sine and cosine values during the simulation. See “Selecting the Twiddle Factor Computation Method” on page 7-399.

This parameter must be set to `Table lookup` for fixed-point signals.

Optimize table for

Select the optimization of the table of sine and cosine values for `Speed` or `Memory`. This parameter is only available when the **Twiddle factor computation** parameter is set to `Table lookup`. See “Optimizing the Table of Trigonometric Values” on page 7-399.

This parameter must be set to `Speed` for fixed-point signals.

Input is in bit-reversed order

Designate the order of the input channel elements. Select when the input is in bit-reversed order, and clear when the input is in linear order. The block yields invalid outputs if you do not set this parameter correctly. See "Input Order" on page 7-400.

Input is conjugate symmetric

Select when the input to the block is both floating point and conjugate symmetric, and you want real-valued outputs. The block output is invalid if you set this parameter when the input is not conjugate symmetric. This parameter cannot be used for fixed-point signals.

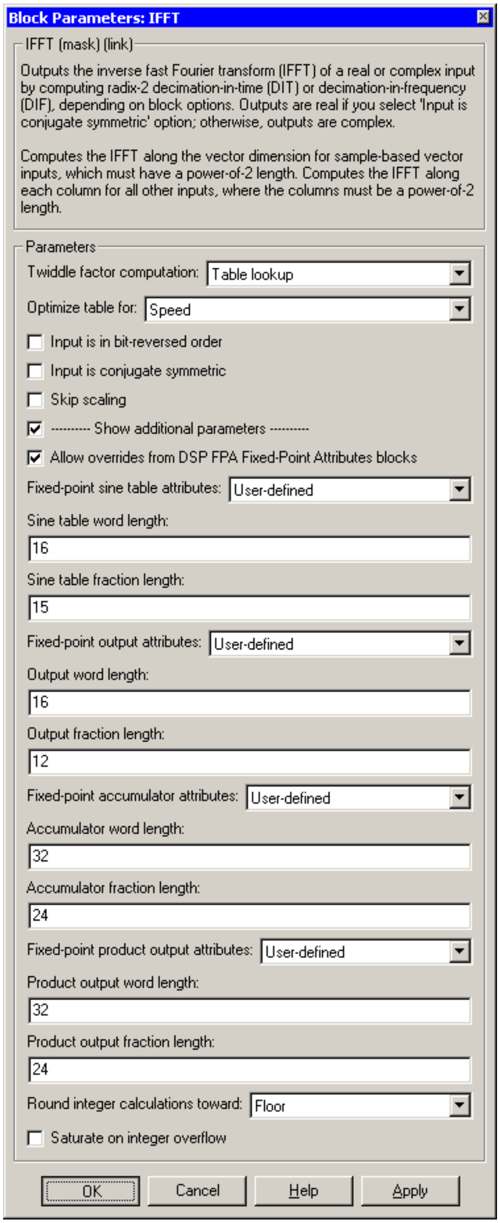
Skip scaling

When you select this check box, no scaling occurs. When this parameter is unselected, scaling does occur:

- For floating-point signals, rather than computing the IDFT, the block computes a scaled version of the IDFT. This scaled version of the IDFT does not include the multiplication factor of 1/M.
- For fixed-point signals, the output of each butterfly of the IFFT is divided by two.

Show additional parameters

If selected, additional parameters specific to implementation of the block become visible as shown.



Block Parameters: IFFT

IFFT (mask) (link)

Outputs the inverse fast Fourier transform (IFFT) of a real or complex input by computing radix-2 decimation-in-time (DIT) or decimation-in-frequency (DIF), depending on block options. Outputs are real if you select 'Input is conjugate symmetric' option; otherwise, outputs are complex.

Computes the IFFT along the vector dimension for sample-based vector inputs, which must have a power-of-2 length. Computes the IFFT along each column for all other inputs, where the columns must be a power-of-2 length.

Parameters

Twiddle factor computation: Table lookup

Optimize table for: Speed

□ Input is in bit-reversed order

□ Input is conjugate symmetric

□ Skip scaling

☑ --------- Show additional parameters ---------

☑ Allow overrides from DSP FPA Fixed-Point Attributes blocks

Fixed-point sine table attributes: User-defined

Sine table word length: 16

Sine table fraction length: 15

Fixed-point output attributes: User-defined

Output word length: 16

Output fraction length: 12

Fixed-point accumulator attributes: User-defined

Accumulator word length: 32

Accumulator fraction length: 24

Fixed-point product output attributes: User-defined

Product output word length: 32

Product output fraction length: 24

Round integer calculations toward: Floor

□ Saturate on integer overflow

Allow overrides from DSP Fixed-Point Attributes blocks

If you select this parameter, fixed-point data types for this block may be set by DSP Fixed-Point Attributes blocks in your model. If this parameter is unselected, the data types are always set by the parameters in the block mask.

Fixed-point sine table attributes

Choose how you will specify the word length and fraction length of the values of the sine table. If you select `Same as input`, the word and fraction lengths of the sine table values are the same as those of the input of the block. If you select `Same as output`, they are the same as those of the output of the block. If you select `User-defined`, the **Fixed-point sine table word length** and **Fixed-point sine table fraction length** parameters become visible.

The sine table values do not obey the **Round integer calculations toward** and **Saturate on integer overflow** parameters; they are always saturated and rounded to `Nearest`.

Fixed-point sine table word length

Specify the word length, in bits, of the sine table values. This parameter is only visible if `User-defined` is specified for the **Fixed-point sine table attributes** parameter.

Fixed-point sine table fraction length

Specify the fraction length, in bits, of the sine table values. This parameter is only visible if `User-defined` is specified for the **Fixed-point sine table attributes** parameter.

Fixed-point output attributes

Choose how you will specify the output word length and fraction length. If you select `Same as input`, these characteristics will match those of the input to the block. If you select `User-defined`, the **Output word length** and **Output fraction length** parameters become visible.

Output word length

Specify the word length, in bits, of the output. This parameter is only visible if `User-defined` is specified for the **Fixed-point output attributes** parameter.

Output fraction length

Specify the fraction length, in bits, of the output. This parameter is only visible if `User-defined` is specified for the **Fixed-point output attributes** parameter.

Fixed-point accumulator attributes

Use this parameter to specify how you would like to designate the accumulator word and fraction lengths. Refer to “Fixed-Point Data Types” on page 7-402 and “Multiplication Data Types” on page 6-15 for illustrations depicting the use of the accumulator data type in this block.

If you select `Same as input`, the accumulator word and fraction lengths are the same as those of the input to the block. If you select `Same as output`, they are the same as those of the output of the block. If you select `User-defined`, the **Accumulator word length** and **Accumulator fraction length** parameters become visible.

Accumulator word length

Specify the word length, in bits, of the accumulator. This parameter is only visible if `User-defined` is specified for the **Fixed-point accumulator attributes** parameter.

Accumulator fraction length

Specify the fraction length, in bits, of the accumulator. This parameter is only visible if `User-defined` is specified for the **Fixed-point accumulator attributes** parameter.

Fixed-point product output attributes

Use this parameter to specify how you would like to designate the product output word and fraction lengths. Refer to “Fixed-Point Data Types” on page 7-402 and “Multiplication Data Types” on page 6-15 for illustrations depicting the use of the product output data type in this block.

If you select `Same as input`, `Same as output`, or `Same as accumulator`, the product output word and fraction lengths are the same as those of the input, output, or accumulator of the block, respectively. If you select `User-defined`, the **Product output word length** and **Product output fraction length** parameters become visible.

Product output word length

Specify the word length, in bits, of the product output. This parameter is only visible if `User-defined` is specified for the **Fixed-point product output attributes** parameter.

Product output fraction length

Specify the fraction length, in bits, of the product output. This parameter is only visible if `User-defined` is specified for the **Fixed-point product output attributes** parameter.

Round integer calculations toward

Select the rounding mode for fixed-point operations. The sine table values do not obey this parameter; they always round to `Nearest`.

Saturate on integer overflow

If selected, overflows saturate. The sine table values do not obey this parameter; they are always saturated.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point
- 8-, 16-, and 32-bit signed integers

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

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

Inherit Complexity

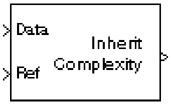
Purpose

Change the complexity of the input to match that of a reference signal

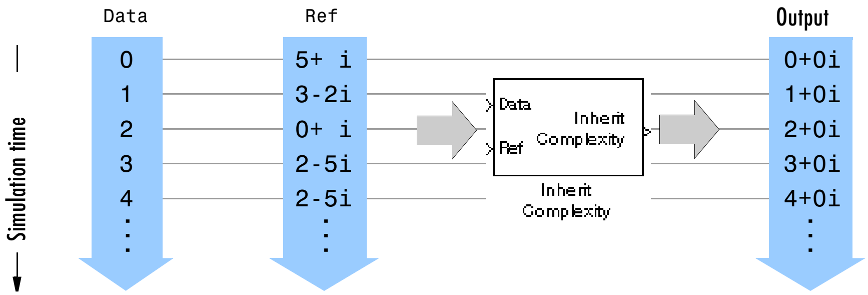
Library

Signal Management / Signal Attributes

Description

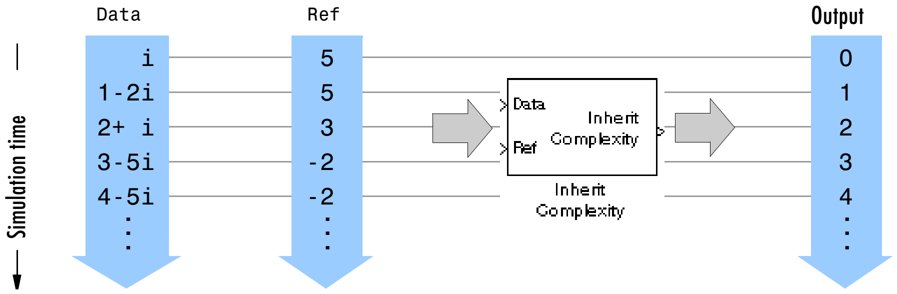


The Inherit Complexity block alters the input data at the `Data` port to match the complexity of the reference input at the `Ref` port. If the `Data` input is real, and the `Ref` input is complex, the block appends a zero-valued imaginary component, `0i`, to each element of the `Data` input.



Data	Ref	Output
0	5+ i	0+0i
1	3-2i	1+0i
2	0+ i	2+0i
3	2-5i	3+0i
4	2-5i	4+0i
⋮	⋮	⋮

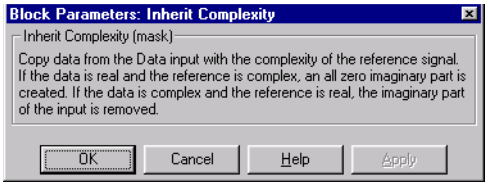
If the `Data` input is complex, and the `Ref` input is real, the block outputs the real component of the `Data` input.



Data	Ref	Output
i	5	0
1-2i	5	1
2+ i	3	2
3-5i	-2	3
4-5i	-2	4
⋮	⋮	⋮

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



Block Parameters: Inherit Complexity

Inherit Complexity (mask)

Copy data from the Data input with the complexity of the reference signal. If the data is real and the reference is complex, an all zero imaginary part is created. If the data is complex and the reference is real, the imaginary part of the input is removed.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point
- Custom data types
- Boolean
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

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

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

Integer Delay

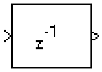
Purpose

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.

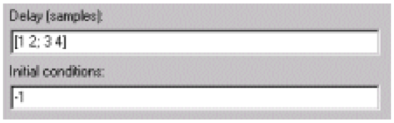
For example, if the input is M-by-1 and v is the matrix `[v(1) v(2) ... v(M)]'`, the first channel is delayed by `v(1)` sample intervals, the second channel is delayed by `v(2)` sample intervals, and so on. Note that when a channel is delayed for Δ sample-time units, the output sample at time t is the 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,



Delay (samples):
[1 2; 3 4]
Initial conditions:
-1

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.

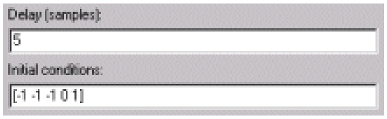
- *Array* of size M-by-N-by-d. In this case, you can set different fixed initial conditions for each element of a sample-based input. This setting is explained further in the *Array* bullet in “Time-Varying Initial Conditions” below.

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.

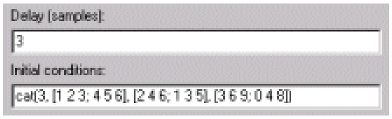
  For a scalar input and the parameters shown below,

  Delay (samples):
  5
  Initial conditions:
  [-1 -1 -1 0 1]

  the block outputs the sequence `-1, -1, -1, 0, 1,...` at the start of the simulation.
- *Array* of dimension M-by-N-by-d, where d is the value specified for the **Delay** parameter (the *maximum* value 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,



Delay (samples):

3

Initial conditions:

cat(3, [1 2 3; 4 5 6], [2 4 6; 1 3 5], [3 6 9; 0 4 8])

the block outputs the matrix sequence

$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix}, \begin{bmatrix}2 & 4 & 6\\1 & 3 & 5\end{bmatrix}, \begin{bmatrix}3 & 6 & 9\\0 & 4 & 8\end{bmatrix}$$

at the start of the simulation. Note that setting **Initial conditions** to an array with the same matrix for each entry implements *constant* initial conditions; a different constant initial condition for each input matrix element (channel).

Initial conditions cannot be specified by full matrices.

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.

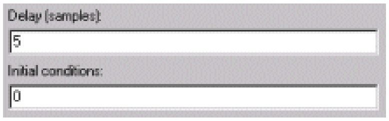
For frame-based inputs, the **Delay** parameter can be a scalar integer by which to equally delay all channels. It can also be a 1-by-N row vector, each element of which serves as the delay for the corresponding channel of the N-channel input. Likewise, it can also be an M-by-1 column vector, each element 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,

  Delay (samples):
  5
  Initial conditions:
  0

  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,

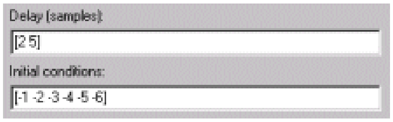
    `D = d + 1`

    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,



Delay (samples):
[2 5]

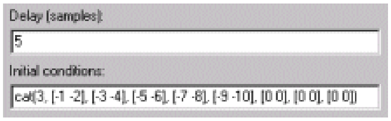
Initial conditions:
[-1 -2 -3 -4 -5 -6]

the block outputs the following sequence of frames at the start of the simulation.

$$\begin{bmatrix} -4 & -1 \\ -5 & -2 \\ 1 & -3 \\ 2 & -4 \end{bmatrix}, \begin{bmatrix} 3 & -5 \\ 4 & 1 \\ 5 & 2 \\ 6 & 3 \end{bmatrix}, \begin{bmatrix} 7 & 4 \\ 8 & 5 \\ 9 & 6 \\ 10 & 7 \end{bmatrix}, \dots$$

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 7-415. In this case, the *k*th entry of each 1-by-N entry in the array corresponds to an initial condition for the *k*th channel of the input matrix. Thus, a 1-by-N-by-D initial conditions input allows you to specify different initial conditions for each channel. For instance, for a two-channel ramp input `[1:100; 1:100]'` with a frame size of 4 and the parameter settings below,



Delay (samples):

5

Initial conditions:

cat(3, [-1 -2], [-3 -4], [-5 -6], [-7 -8], [-9 -10], [0 0], [0 0], [0 0])

the block outputs the following sequence of frames at the start of the simulation.

$$\begin{bmatrix} -1 & -2 \\ -3 & -4 \\ -5 & -6 \\ -7 & -8 \end{bmatrix}, \begin{bmatrix} -9 & -10 \\ 1 & 1 \\ 2 & 2 \\ 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 \\ 5 & 5 \\ 6 & 6 \\ 7 & 7 \end{bmatrix}, \ldots$$

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.

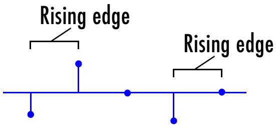
Initial conditions cannot be specified by full matrices.

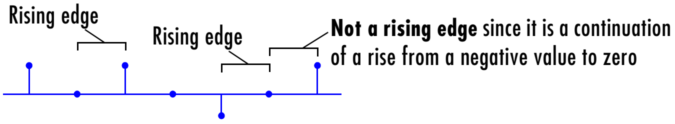
Resetting the Delay

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.

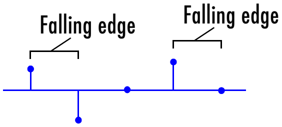
The reset event is specified by the **Reset port** parameter, and can be one of the following:

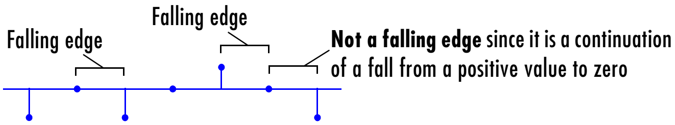
- `None` disables the `Rst` port.
- `Rising edge` — Triggers a reset operation when the `Rst` input does one of the following:
  - Rises from a negative value to a positive value or zero
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)




Rising edge
Rising edge
Not a rising edge since it is a continuation of a rise from a negative value to zero

- 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)




Falling edge
Falling edge
Not a falling edge since it is a continuation of a fall from a positive value to zero

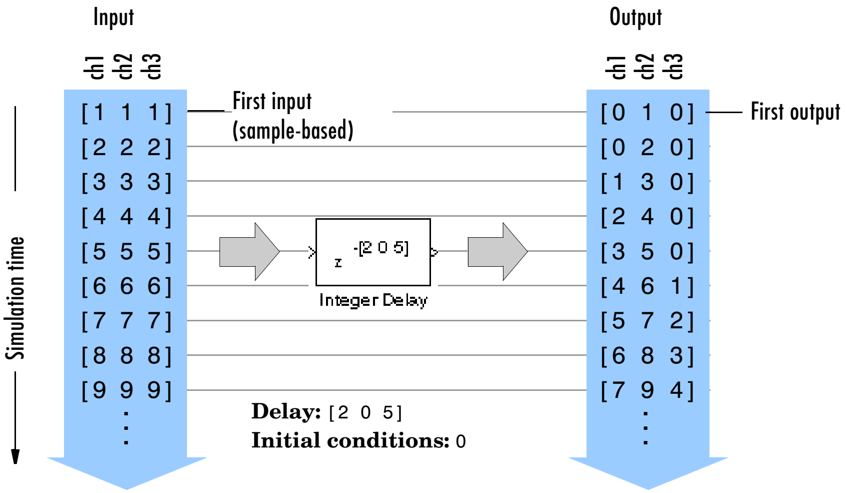
- Either edge — Triggers a reset operation when the Rst input is a Rising edge or Falling edge (as described above).
- Non-zero sample — Triggers a reset operation at each sample time that the Rst input is not zero.

Note When running simulations in the Simulink MultiTasking mode, sample-based reset signals have a one-sample latency, and frame-based reset signals have one frame of latency. Thus, there is a one-sample or one-frame delay between the time the block detects a reset event, and when it applies the reset. For more information on latency and the Simulink tasking modes, see

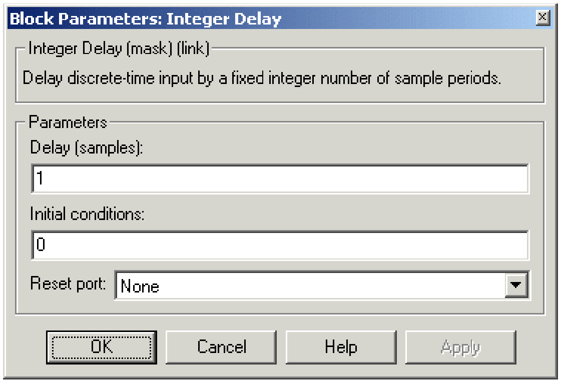
"Excess Algorithmic Delay (Tasking Latency)" on page 2-98 and the topic on The Simulation Parameters Dialog Box in the Simulink documentation.

Examples

The dspafxr demo illustrates an audio reverberation system built around the Integer Delay block.



Dialog Box



Block Parameters: Integer Delay

Integer Delay (mask) (link)

Delay discrete-time input by a fixed integer number of sample periods.

Parameters

Delay (samples): 1

Initial conditions: 0

Reset port: None

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 7-417.

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

See Also

Unit Delay	Simulink
Variable Fractional Delay	DSP Blockset
Variable Integer Delay	DSP Blockset

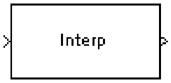
Purpose

Interpolate values of real input samples

Library

Signal Operations

Description



The Interpolation block interpolates each channel of discrete, real, inputs using linear or FIR interpolation. The input can be a sample or frame based vector or matrix. The output is a vector or matrix of the interpolated values, and has the same frame status and frame rate as the input.

You must specify the *interpolation points* (times at which to interpolate values) in an *interpolation vector*, I_n. An entry of 1 in I_n refers to the first sample of the input, an entry of 2.5 refers to the sample half-way between the second and third input sample, and so on. I_n must have the same frame status and frame rate as the input, and can be a length-P row or column vector, where P is usually any positive integer.

Usually, the block applies the vector I_n to each column of an input matrix, or to each input vector. You can set the block to either apply the *same* interpolation vector for all input vectors or matrices (*static* interpolation points), or use a *different* interpolation vector for each input vector or matrix (*time-varying* interpolation points).

For more information, see other sections of this reference page.

Sections of This Reference Page

- "Specifying Static Interpolation Points" on page 7-421
- "Specifying Time-Varying Interpolation Points" on page 7-422
- "How the Block Applies Interpolation Vectors to Inputs" on page 7-422
- "Handling Out-of-Range Interpolation Points" on page 7-424
- "Linear Interpolation Mode" on page 7-425
- "FIR Interpolation Mode" on page 7-426
- "Dialog Box" on page 7-427
- "Supported Data Types" on page 7-428

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 7-422.

Specifying Time-Varying Interpolation Points

To supply the block with time-varying interpolation vectors (where the block uses a different interpolation vector for each input vector or matrix), do the following:

1. Set the **Source of interpolation points** parameter to `Input port`, which activates a block input port, In, for the interpolation points.
2. Generate a signal of interpolation vectors with the *same frame status* and *same frame rate* as the input signal, and supply it to the input port for interpolation points. To learn about interpolation vectors, see "How the Block Applies Interpolation Vectors to Inputs" on page 7-422.

How the Block Applies Interpolation Vectors to Inputs

The interpolation vector I_n represents the points in time at which to interpolate values of the input signal. An entry of 1 in I_n refers to the first sample of the input, an entry of 2.5 refers to the sample half-way between the second and third input sample, and so on. In most cases, the vector I_n can be of any length.

Depending on the dimension and frame status of the input and the dimension of I_n, the block usually applies I_n to the input in one of the following ways:

- Applies the vector I_n to each channel of a matrix input, resulting in a matrix output.
- Applies the vector I_n to each input vector (as if the input vector were a single channel), resulting in a vector output with the same orientation as the input (row or column).

The following tables summarize how the block applies the vector I_n to all the possible types of sample- and frame-based inputs, and show the resulting output dimensions. (The block applies both static and time-varying interpolation vectors to the input signal in the same way).

How Block Applies Interpolation Vectors to Frame-Based Inputs

Frame-Based Input Dimensions	Dimensions of Interpolation Vector I_n (P is a positive integer)	How Block Applies I_n to Input	Frame-Based Output Dimensions
M-by-N matrix	P-by-1 column	Applies I_n to each input column	P-by-N matrix
	1-by-N row	Applies each column of I_n (each element of I_n) to the corresponding columns of the input	1-by-N row
M-by-1 column	P-by-1 column	Applies I_n to the input column	P-by-1 column
	1-by-P row (block treats as a column)	Applies I_n to the input column	P-by-1 column
1-by-N row (not recommended)	P-by-1 column	not applicable	P-by-N matrix where each row is a copy of the input vector
	1-by-P row	not applicable	1-by-N row, a copy of the input vector

How Block Applies Interpolation Vectors to Sample-Based Inputs

Sample-Based Input Dimensions	Dimensions of Interpolation Vector I_n (P is any positive integer)	How Block Applies I_n to Input	Sample-Based Output Dimensions
M-by-N matrix	P-by-1 column	Applies I_n to each input column	P-by-N matrix
	1-by-P row (block treats as a column)	Applies I_n to each input column	P-by-N matrix
M-by-1 column	P-by-1 column	Applies I_n to the input column	P-by-1 column
	1-by-P row (block treats as a column)	Applies I_n to the input column	P-by-1 column
1-by-N row	P-by-1 column (block treats as a row)	Applies I_n to the input row	1-by-P row
	1-by-P row	Applies I_n to the input row	1-by-P row

Handling Out-of-Range Interpolation Points

The *valid range* of the values in the interpolation vector I_n is from 1 to the number of samples in each channel of the input. For instance, given a length-5 input vector `D`, all entries of I_n must range from 1 to 5. I_n cannot contain entries such as 7 or -9, since there is no 7th or -9th entry in `D`.

The **Out of range interpolation points** parameter sets how the block handles interpolation points that are not within the valid range, and has the following settings:

- `Clip` — The block replaces any out-of-range values in I_n with the closest value in the valid range (from 1 to the number of input samples), and then proceeds with computations using the clipped version of I_n.
- `Clip and warn` — In addition to `Clip`, the block issues a warning at the MATLAB command line every time clipping occurs.

- `Error` — When the block encounters an out-of-range value in I_n, the simulation stops and the block issues an error at the MATLAB command line.

Example of Clipping . Suppose the block is set to clip out-of-range interpolation points, and gets the following input vector and interpolation points:

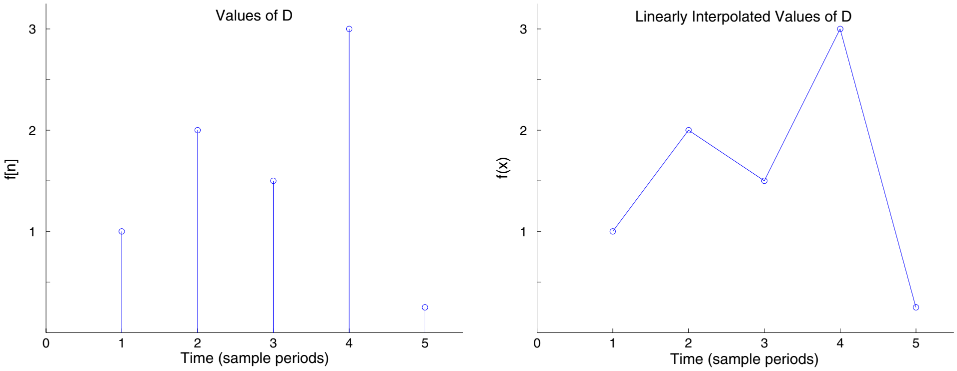
- `D = [11, 22, 33, 44]'`
- I_n `= [10, 2.6, -3]'`

Since `D` has four samples, valid interpolation points range from 1 to 4. The block clips the interpolation point 10 to 4 and the point -3 to 1, resulting in the clipped interpolation vector $I_{nclipped}$ `= [4, 2.6, 1]'`.

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.

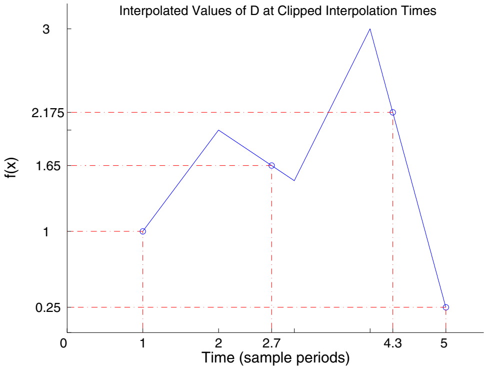
For instance, if the input signal `D = [1, 2, 1.5, 3, 0.25]'`, the following left-hand plot shows the samples in `D`, and the right-hand plot shows the linearly interpolated values between the samples in `D`.



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]'`.



`D = [1, 2, 1.5, 3, 0.25]'`

I_n `= [-4, 2.7, 4.3, 10]'`

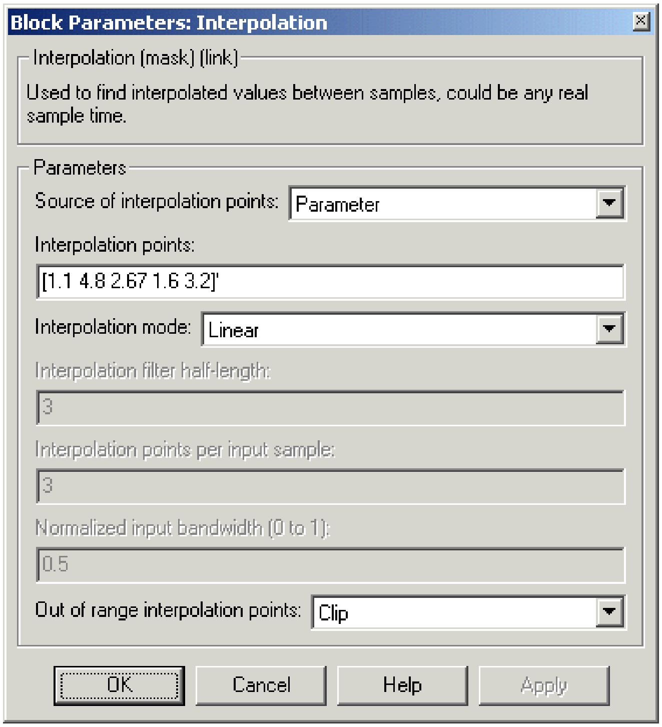
The valid time range is from 1 to 5 sample periods, so -4 is clipped to 1, and 10 is clipped to 5.

Clipped I_n `= [1, 2.7, 4.3, 5]'`

FIR Interpolation Mode

When **Interpolation Mode** is set to `FIR`, the block interpolates data values using an FIR interpolation filter, specified by various block parameters. See "FIR Interpolation Mode" on page 7-800 in the Variable Fractional Delay block reference for more information.

Dialog Box



Block Parameters: Interpolation

Interpolation (mask) (link)

Used to find interpolated values between samples, could be any real sample time.

Parameters

Source of interpolation points: Parameter

Interpolation points: [1.1 4.8 2.67 1.6 3.2]'

Interpolation mode: Linear

Interpolation filter half-length: 3

Interpolation points per input sample: 3

Normalized input bandwidth (0 to 1): 0.5

Out of range interpolation points: Clip

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 7-421 and “Specifying Time-Varying Interpolation Points” on page 7-422. Nontunable.

Interpolation points

The vector I_n of points in time at which to interpolate the input signal. An entry of 1 in I_n refers to the first sample of the input, an entry of 2.5 refers to the sample half-way between the second and third input sample, and so on. See “How the Block Applies Interpolation Vectors to Inputs” on page 7-422. Tunable.

Interpolation mode

Sets the block to interpolate by either linear or FIR interpolation. For more information, see “Linear Interpolation Mode” on page 7-425 and “FIR Interpolation Mode” on page 7-426. Nontunable.

Interpolator filter half-length

Half the length of the FIR interpolation filter. For more information, see “FIR Interpolation Mode” on page 7-426. Nontunable.

Interpolation points per input sample

The number Q, where the FIR interpolation filter uses the nearest 2*Q points in the signal to interpolate the value at an interpolation point. 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 7-426. and “Linear Interpolation Mode” on page 7-425. Nontunable.

Normalized input bandwidth (0 to 1)

The bandwidth of the input divided by Fs/2 (half the input sample frequency). For more information, see “FIR Interpolation Mode” on page 7-426. Nontunable.

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 7-424. Nontunable.

Supported Data Types

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

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

Kalman Adaptive Filter

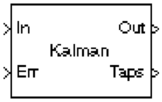
Purpose

Compute filter estimates for an input using the Kalman adaptive filter algorithm

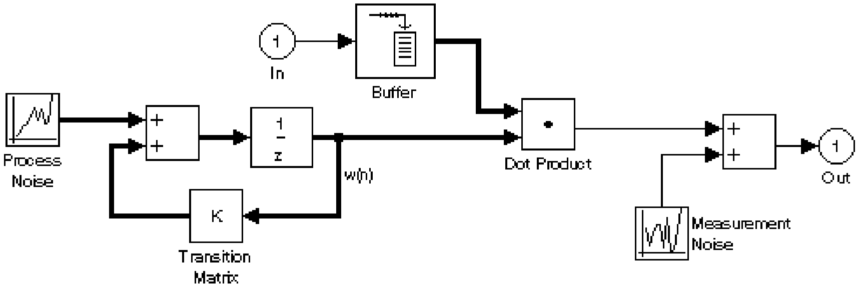
Library

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 dynamic system.



The Kalman filter assumes that there are no deterministic changes to the filter taps over time (that is, the transition matrix is identity), and that the only observable output from the system is the filter output with additive noise. The corresponding Kalman filter is expressed in matrix form as

$$
\begin{aligned}
g(n) &= \frac{K(n-1)u(n)}{u^H(n)K(n-1)u(n) + Q_M} \\
y(n) &= u^H(n)\hat{w}(n) \\
e(n) &= d(n) - y(n) \\
\hat{w}(n+1) &= \hat{w}(n) + e(n)g(n) \\
K(n) &= K(n-1) - g(n)u^H(n)K(n-1) + Q_P
\end{aligned}
$$

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.

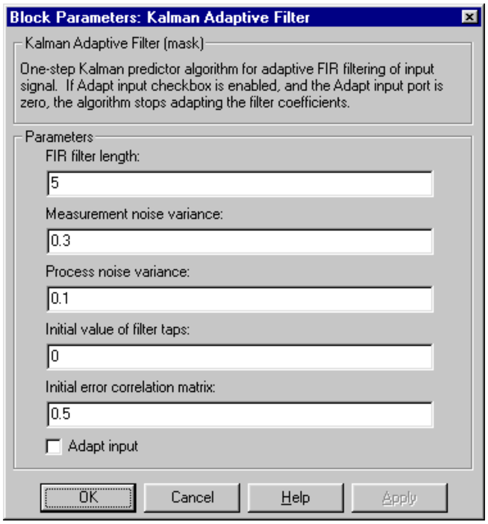
Block Ports	Corresponding Variables
`In`	u, the scalar input, which is internally buffered into the vector $u(n)$
`Out`	$y(n)$, the filtered scalar output
`Err`	$e(n)$, the scalar estimation error
`Taps`	$\hat{w}(n)$, the vector of filter-tap estimates

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



Block Parameters: Kalman Adaptive Filter

Kalman Adaptive Filter (mask)

One-step Kalman predictor algorithm for adaptive FIR filtering of input signal. If Adapt input checkbox is enabled, and the Adapt input port is zero, the algorithm stops adapting the filter coefficients.

Parameters

FIR filter length: 5

Measurement noise variance: 0.3

Process noise variance: 0.1

Initial value of filter taps: 0

Initial error correlation matrix: 0.5

□ Adapt input

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 your data types to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-2.

See Also

LMS Adaptive Filter	DSP Blockset
RLS Adaptive Filter	DSP Blockset

See “Adaptive Filters” on page 3-46 for related information.

LDL Factorization

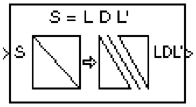
Purpose

Factor a square Hermitian positive definite matrix into lower, upper, and diagonal components

Library

Math Functions / Matrices and Linear Algebra / Matrix Factorizations

Description

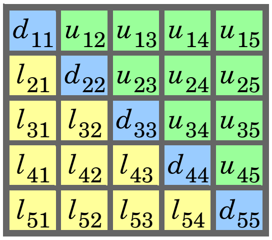


The LDL Factorization block uniquely factors the square Hermitian positive definite input matrix S as

$$S = LDL^*$$

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.



d_{11}	u_{12}	u_{13}	u_{14}	u_{15}
l_{21}	d_{22}	u_{23}	u_{24}	u_{25}
l_{31}	l_{32}	d_{33}	u_{34}	u_{35}
l_{41}	l_{42}	l_{43}	d_{44}	u_{45}
l_{51}	l_{52}	l_{53}	l_{54}	d_{55}

$$u_{ij} = l_{ji}^*$$

LDL factorization requires half the computation of Gaussian elimination (LU decomposition), and is always stable. It is more efficient that Cholesky factorization because it avoids computing the square roots of the diagonal elements.

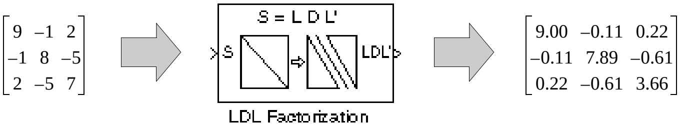
The algorithm requires that the input be square and Hermitian positive definite. When the input is not positive definite, the block reacts with the behavior specified by the **Non-positive definite input** parameter.

The following options are available:

- `Ignore` — Proceed with the computation and *do not* issue an alert. The output is *not* a valid factorization. A partial factorization will be present in the upper left corner of the output.
- `Warning` — Display a warning message in the MATLAB Command Window, and continue the simulation. The output is *not* a valid factorization. A partial factorization will be present in the upper left corner of the output.
- `Error` — Display an error dialog box and terminate the simulation.

Examples

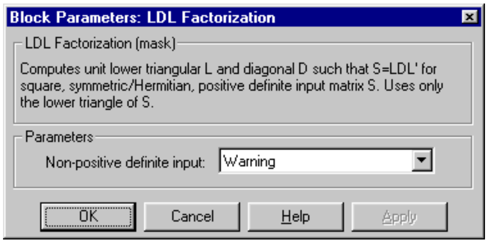
LDL decomposition of a 3-by-3 Hermitian positive definite matrix:



$$\begin{bmatrix} 9 & -1 & 2 \\ -1 & 8 & -5 \\ 2 & -5 & 7 \end{bmatrix} \Rightarrow \begin{bmatrix} 9.00 & -0.11 & 0.22 \\ -0.11 & 7.89 & -0.61 \\ 0.22 & -0.61 & 3.66 \end{bmatrix}$$

$$L = \begin{bmatrix} 1 & 0 & 0 \\ -0.11 & 1 & 0 \\ 0.22 & -0.61 & 1 \end{bmatrix} \qquad D = \begin{bmatrix} 9.00 & 0 & 0 \\ 0 & 7.89 & 0 \\ 0 & 0 & 3.66 \end{bmatrix} \qquad L' = \begin{bmatrix} 1 & -0.11 & 0.22 \\ 0 & 1 & -0.61 \\ 0 & 0 & 1 \end{bmatrix}$$

Dialog Box



Block Parameters: LDL Factorization

LDL Factorization (mask)

Computes unit lower triangular L and diagonal D such that S=LDL' for square, symmetric/Hermitian, positive definite input matrix S. Uses only the lower triangle of S.

Parameters

Non-positive definite input: Warning

Non-positive definite input

Response to nonpositive definite matrix inputs.

References

Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

Supported Data Types

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

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

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 5-8 for related information.

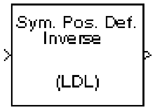
Purpose

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.

$$S^{-1} = (LDL^*)^{-1}$$

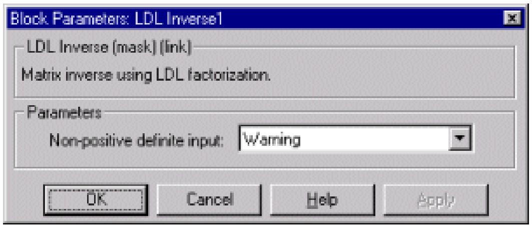
L is a lower triangular square matrix with unity diagonal elements, D is a diagonal matrix, and L^* is the Hermitian (complex conjugate) transpose of L. Only the diagonal and lower triangle of the input matrix are used, and any imaginary component of the diagonal entries is disregarded. The output is 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



Block Parameters: LDL Inverse1

LDL Inverse (mask) (link)

Matrix inverse using LDL factorization.

Parameters

Non-positive definite input: Warning

Non-positive definite input

Response to nonpositive definite matrix inputs.

References

Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

Supported Data Types

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

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

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 5-9 for related information.

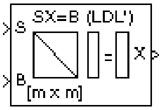
Purpose

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 LDL Solver block solves the linear system S*X*=B by applying LDL factorization to the matrix at the S port, which must be square (M-by-M) and Hermitian positive definite. Only the diagonal and lower triangle of the matrix are used, and any imaginary component of the diagonal entries is disregarded. The input to the B port is the right side M-by-N matrix, B. The output is the unique solution of the equations, M-by-N matrix *X*, and is always sample based.

A length-M 1-D vector input for right side B is treated as an M-by-1 matrix.

When the input is not positive definite, the block reacts with the behavior specified by the **Non-positive definite input** parameter. The following options are available:

- `Ignore` — Proceed with the computation and *do not* issue an alert. The output is *not* a valid solution.
- `Warning` — Proceed with the computation and display a warning message in the MATLAB Command Window. The output is *not* a valid solution.
- `Error` — Display an error dialog box and terminate the simulation.

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

   $$Z = L^*X$$

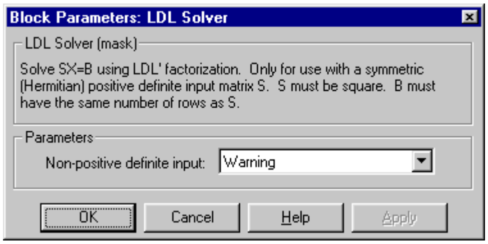
3. Solve one diagonal and two triangular systems.

   $$LY = B$$

   $$DZ = Y$$

   $$L^*X = Z$$

Dialog Box



Block Parameters: LDL Solver

LDL Solver (mask)

Solve SX=B using LDL' factorization. Only for use with a symmetric (Hermitian) positive definite input matrix S. S must be square. B must have the same number of rows as S.

Parameters

Non-positive definite input: Warning

Non-positive definite input

Response to nonpositive definite matrix inputs.

Supported Data Types

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

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

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 5-6 for related information.

Least Squares Polynomial Fit

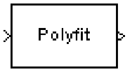
Purpose

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 *n*th 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 *i*th 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, $x_1, x_2, \ldots, x_{\mathrm{M}}$, 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.

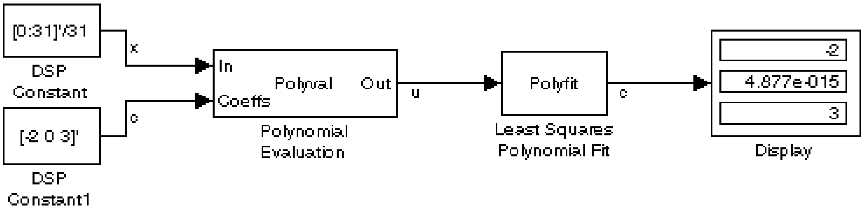
Each column of the (*n*+1)-by-N output matrix, *c*, represents a set of *n*+1 coefficients describing the best-fit polynomial for the corresponding column of the input. The coefficients in each column are arranged in order of descending exponents, $c_1, c_2, \ldots, c_{n+1}$. The output is always sample based.

Examples

In the model below, the Polynomial Evaluation block uses the second-order polynomial

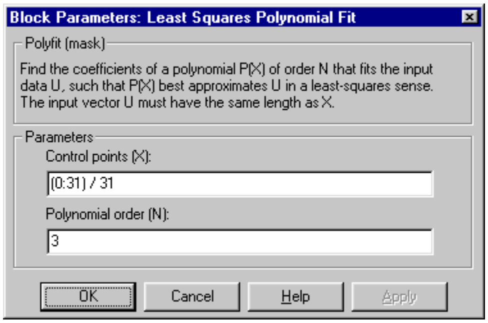
$$y = -2u^2 + 3$$

to generate four values of dependent variable *y* from four values of independent variable *u*, received at the top port. The polynomial coefficients are supplied in the vector `[-2 0 3]` at the bottom port. Note that the coefficient of the first-order term is zero.



The **Control points** parameter of the Least Squares Polynomial Fit block is configured with the same four values of independent variable *u* that are used as input to the Polynomial Evaluation block, `[1 2 3 4]`. The Least Squares Polynomial Fit block uses these values together with the input values of dependent variable *y* to reconstruct the original polynomial coefficients.

Dialog Box


Block Parameters: Least Squares Polynomial Fit

Polyfit (mask)

Find the coefficients of a polynomial P(X) of order N that fits the input data U, such that P(X) best approximates U in a least-squares sense. The input vector U must have the same length as X.

Parameters

Control points (X): (0:31) / 31

Polynomial order (N): 3

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.

Polynomial order

The order, *n*, of the polynomial to be used in constructing the best fit. The number of coefficients is *n*+1.

Supported Data Types

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

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

See Also

Detrend	DSP Blockset
Polynomial Evaluation	DSP Blockset
Polynomial Stability Test	DSP Blockset
`polyfit`	MATLAB

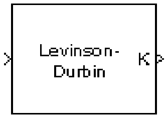
Purpose

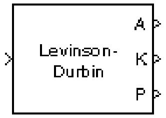
Solve a linear system of equations using Levinson-Durbin recursion

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 the **If the value of lag 0 is zero, A=[1 zeros], K=[zeros], P=0** check box 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.

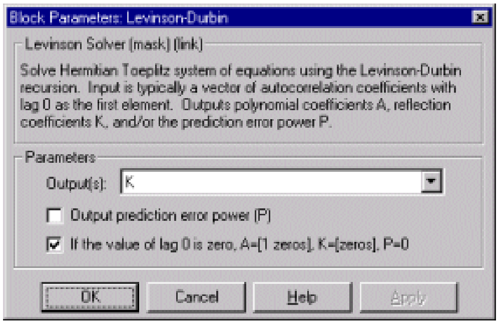
$$\underset{\{a_i\}}{min} \quad E\left[\left|x_n - \sum_{i=1}^{N} a_i x_{n-i}\right|^2\right]$$

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


Block Parameters: Levinson-Durbin
Levinson Solver (mask) (link)
Parameters
OK
Cancel
Help
Apply

Solve Hermitian Toeplitz system of equations using the Levinson-Durbin recursion. Input is typically a vector of autocorrelation coefficients with lag 0 as the first element. Outputs polynomial coefficients A, reflection coefficients K, and/or the prediction error power P.

Output(s): K

□ Output prediction error power (P)

☑ If the value of lag 0 is zero, A=[1 zeros], K=[zeros], P=0

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 your data types to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-2.

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 5-6 for related information.

LMS Adaptive Filter

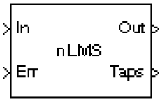
Purpose

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.

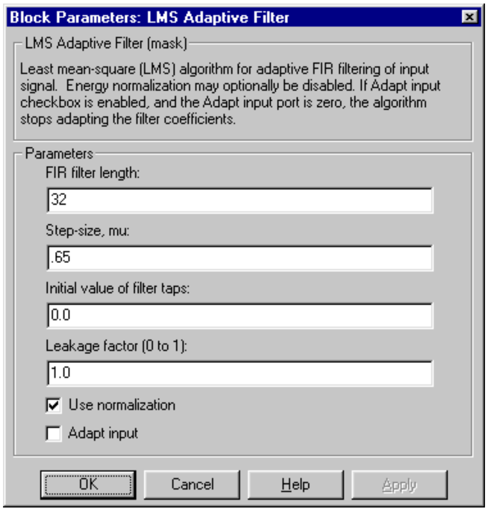
The **FIR filter length** parameter specifies the length of the filter that the LMS algorithm estimates. The **Step size** parameter corresponds to μ in the equations. Typically, for convergence in the mean square, μ must be greater than 0 and less than 2. The **Initial value of filter taps** specifies the initial value $\hat{w}(0)$ as a vector, or as a scalar to be repeated for all vector elements. The **Leakage factor** specifies the 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



Block Parameters: LMS Adaptive Filter

LMS Adaptive Filter (mask)

Least mean-square (LMS) algorithm for adaptive FIR filtering of input signal. Energy normalization may optionally be disabled. If Adapt input checkbox is enabled, and the Adapt input port is zero, the algorithm stops adapting the filter coefficients.

Parameters

FIR filter length: 32

Step-size, mu: .65

Initial value of filter taps: 0.0

Leakage factor (0 to 1): 1.0

☑ Use normalization

□ Adapt input

FIR filter length

The length of the FIR filter.

Step-size

The step-size, usually in the range (0, 2). Tunable.

Initial value of filter taps

The initial FIR filter coefficients.

Leakage factor

The leakage factor, in the range [0, 1]. Tunable.

Use normalization

Select this check box to compute the filter-tap estimate using the normalized equations.

Adapt input

Enables the `Adapt` port if selected.

References

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

Supported Data Types

- Double-precision floating point

- Single-precision floating point

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

See Also

Kalman Adaptive Filter	DSP Blockset
RLS Adaptive Filter	DSP Blockset

See “Adaptive Filters” on page 3-46 for related information.

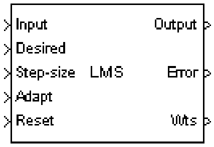
Purpose

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

Library

Filtering / Adaptive Filters

Description



LMS Filter

The LMS Filter block is capable of implementing an adaptive FIR filter using five different algorithms. The block estimates the filter weights, or coefficients, needed to convert the input signal into the desired signal. Connect the signal you want to filter to the Input port. This input can be a sample-based or frame-based signal. Connect the signal you want to model to the Desired port. The desired signal must have the same data type, signal type (sample or frame based), and dimensions as the input signal. The Output port outputs the filtered input signal, which is the estimate of the desired signal. The output of the Output port can be sample or frame based. The Error port outputs the result of subtracting the output signal from the desired signal.

If you select `LMS` for the **Algorithm** parameter, the block calculates the filter weights using the least mean-square (LMS) algorithm. This algorithm is defined by the following equations.

$$\begin{aligned} y(n) &= \mathbf{w}^T(n-1)\mathbf{u}(n) \\ e(n) &= d(n) - y(n) \\ \mathbf{w}(n) &= \mathbf{w}(n-1) + f(\mathbf{u}(n), e(n), \mu) \end{aligned}$$

The weight update function, for the LMS algorithm, is defined as

$$f(\mathbf{u}(n), e(n), \mu) = \mu e(n)\mathbf{u}^*(n)$$

The variables are as follows.

Variable	Description
n	The current time index
$\mathbf{u}(n)$	The vector of buffered input samples at step n
$\hat{\mathbf{w}}(n)$	The vector of filter weight estimates at step n
$y(n)$	The filtered output at step n

Variable	Description
$e(n)$	The estimation error at step n
$d(n)$	The desired response at step n
μ	The adaptation step size

If you select `Normalized LMS` for the **Algorithm** parameter, the block calculates the filter weights using the normalized LMS algorithm. This algorithm is defined by the following equations.

$$\begin{aligned} y(n) &= \mathbf{w}(n-1)\mathbf{u}(n) \\ e(n) &= d(n) - y(n) \\ \mathbf{w}(n) &= \mathbf{w}(n-1) + f(\mathbf{u}(n), e(n), \mu) \end{aligned}$$

The weight update function, for the normalized LMS algorithm, is defined as

$$f(\mathbf{u}(n), e(n), \mu) = \mu e(n)\frac{\mathbf{u}^*(n)}{a + \mathbf{u}^H(n)\mathbf{u}(n)}$$

To overcome potential numerical instability in the update of the weights, a small positive constant (a = 1e-10) has been added in the denominator.

If you select `Sign-Error LMS` for the **Algorithm** parameter, the block calculates the filter weights using the LMS algorithm equations. However, each time the block updates the weights, it replaces the error term, $e(n)$, with +1 if the error term is positive or -1 if the error term is negative.

If you select `Sign-Data LMS` for the **Algorithm** parameter, the block calculates the filter weights using the LMS algorithm equations. However, each time the block updates the weights, it replaces each sample of the input vector, $\mathbf{u}(n)$, with +1 if the input sample is positive or -1 is the input sample is negative.

If you select `Sign-Sign LMS` for the **Algorithm** parameter, the block calculates the filter weights using the LMS algorithm equations. However, each time the block updates the weights, it replaces the error term, $e(n)$, with +1 if the error term is positive or -1 is the error term is negative. It also replaces each sample of the input vector, $\mathbf{u}(n)$, with +1 if the input sample is positive or -1 is the input sample is negative.

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

The **Step-size (mu)** parameter corresponds to μ in the equations. For convergence of the normalized LMS equations, 0<μ<2. You can either specify a step size using the input port, Step-size, or enter a value in the **Block Parameters: LMS Filter** dialog box.

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

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

If you select `LMS` from the **Algorithm** list, the weight update function in the above equation is the LMS weight update function. If you select `Normalized LMS` from the **Algorithm** list, the weight update function in the above equation is the normalized LMS weight update function.

Enter the initial filter weights, $\hat{\mathbf{w}}(0)$, as a vector or a scalar in the **Initial value of filter weights** text box. If you enter a scalar, the block uses the scalar value to create a vector of filter weights. This vector has length equal to the filter length and all of its values are equal to the scalar value

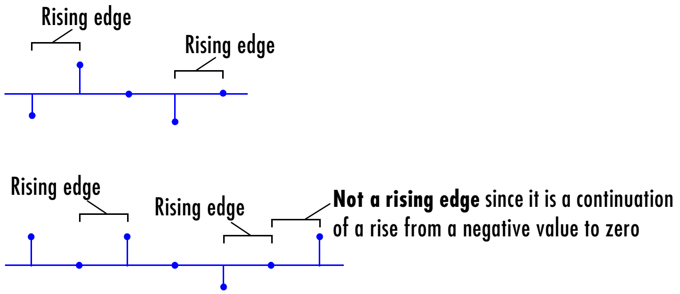
If you select the **Enable/disable adaptation via input port** check box, an Adapt port appears on the block. When the input to this port is nonzero, the block continuously updates the filter weights. When the input to this port is zero, the filter weights remain at their current values.

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

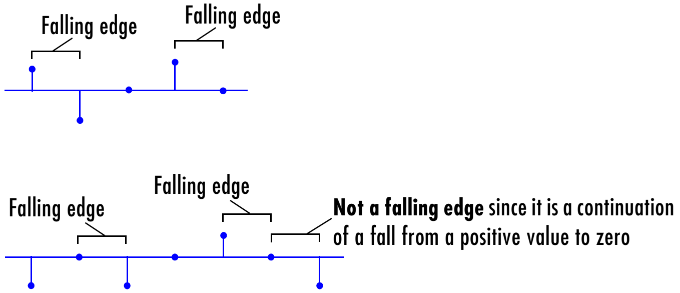
From the **Reset input** list, select `None` to disable the Reset port. To enable the Reset port, select one of the following from the **Reset input** list:

- `Rising edge` — Triggers a reset operation when the Reset input does one of the following:
  - Rises from a negative value to a positive value or zero

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


Rising edge
Rising edge
Rising edge
Rising edge
Not a rising edge since it is a continuation of a rise from a negative value to zero

- `Falling edge` — Triggers a reset operation when the Reset input does one of the following:
  - Falls from a positive value to a negative value or zero
  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero (see the following figure)

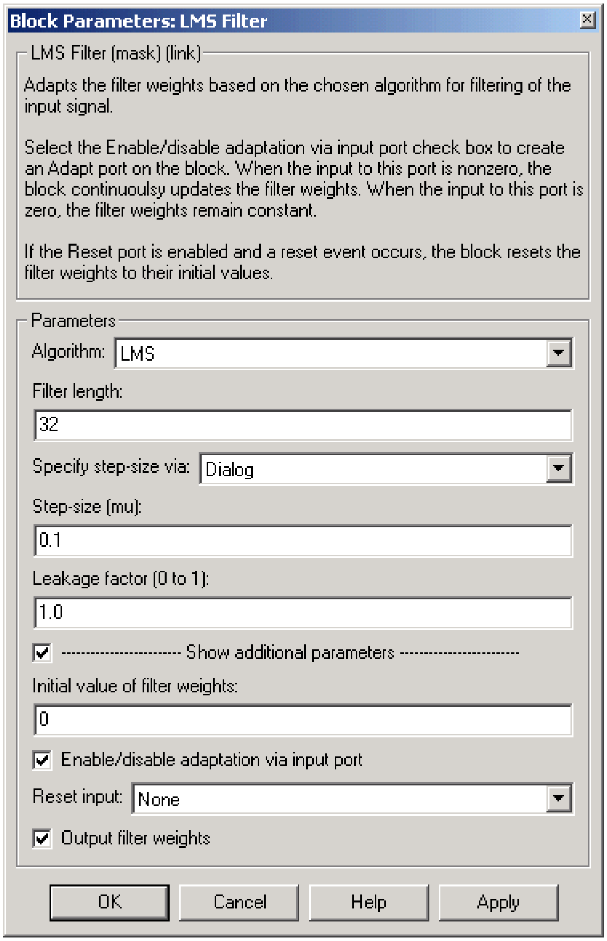

Falling edge
Falling edge
Falling edge
Falling edge
Not a falling edge since it is a continuation of a fall from a positive value to zero

- `Either edge` — Triggers a reset operation when the Reset input is a `Rising edge` or `Falling edge` (as described above)
- `Non-zero sample` — Triggers a reset operation at each sample time that the Reset input is not zero

Note When running simulations in the Simulink `MultiTasking` mode, sample-based reset signals have a one-sample latency, and frame-based reset signals have one frame of latency. Thus, there is a one-sample or one-frame delay between the time the block detects a reset event, and when it applies the reset. For more information on latency and the Simulink tasking modes, see “Excess Algorithmic Delay (Tasking Latency)” on page 2-98 and the topic called “The Simulation Parameters Dialog Box” in the Simulink documentation.

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

Dialog Box



Block Parameters: LMS Filter

LMS Filter (mask) (link)

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

Select the Enable/disable adaptation via input port check box to create an Adapt port on the block. When the input to this port is nonzero, the block continuoulsy updates the filter weights. When the input to this port is zero, the filter weights remain constant.

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

Parameters

Algorithm: LMS

Filter length: 32

Specify step-size via: Dialog

Step-size (mu): 0.1

Leakage factor (0 to 1): 1.0

☑ ------------------------ Show additional parameters ------------------------

Initial value of filter weights: 0

☑ Enable/disable adaptation via input port

Reset input: None

☑ Output filter weights

Algorithm

Choose the algorithm used to calculate the filter weights.

Filter length

Enter the length of the FIR filter weights vector.

Specify step-size via

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

Step-size (mu)

Enter the step-size. Tunable.

Leakage factor (0 to 1)

Enter the leakage factor, $0 < 1 - \mu\alpha \leq 1$. Tunable.

Initial value of filter weights

Specify the initial values of the FIR filter weights.

Enable/disable adaptation via input port

Select this check box to enable the Adapt input port.

Reset input

Select this check box to enable the Reset input port.

Output filter weights

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

References

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

Supported Data Types

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

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

See Also

Kalman Adaptive Filter	DSP Blockset
RLS Filter	DSP Blockset
Block LMS Filter	DSP Blockset
Fast Block LMS Filter	DSP Blockset

See “Adaptive Filters” on page 3-46 for related information.

LPC to LSF/LSP Conversion

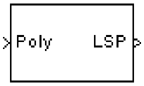
Purpose

Convert linear prediction coefficients (LPCs) to line spectral pairs (LSPs) or line spectral frequencies (LSFs)

Library

Estimation / Linear Prediction

Description



The LPC to LSF/LSP Conversion block takes a vector of linear prediction coefficients (LPCs) and converts it to a vector of line spectral pairs (LSPs) or line spectral frequencies (LSFs). When converting LPCs to LSFs, the block outputs match those of the `poly2lsf` function.

The input LPCs, $1, a_1, a_2, \ldots, a_m$, must be the denominator of the transfer function of a stable all-pole filter with the form given in the first equation of “Requirements for Getting Valid Outputs” on page 7-461. 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 7-461 — 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 7-462 — 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 7-462 — 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 7-463 — Valid input frame statuses, sizes, and dimensions, and those of the corresponding output
- “Handling and Recognizing Invalid Inputs and Outputs” on page 7-465 — How to set block parameters for handling invalid inputs and outputs
- “LSF and LSP Computation Method: Chebyshev Polynomial Method for Root Finding” on page 7-467 — Description and diagram of the block’s root finding method

- "Root Finding Method Limitations: Failure to Find Roots" on page 7-470 — Description and diagram of how the block's root finding method can fail if parameters are not set properly
- "Dialog Box" on page 7-472 — A summary of the block parameters
- "Supported Data Types" on page 7-474 — Supported data types and a link to how to convert data types
- "See Also" on page 7-474 — 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, $1, a_1, a_2, \dots, a_m$, 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.

$$H(z) = \frac{1}{1 + a_1 z^{-1} + a_2 z^{-2} + \dots + a_m z^{-m}}$$

- The **Root finding coarse grid points** parameter value must be large enough so that the block can find all the LSP or LSF values. (The output LSFs and LSPs are roots of polynomials related to the input LPC polynomial; the block looks for these roots to produce the output. For details, see "LSF and LSP Computation Method: Chebyshev Polynomial Method for Root Finding" on page 7-467.) 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 7-470.

To learn about recognizing invalid inputs and outputs and parameters for dealing with them, see "Handling and Recognizing Invalid Inputs and Outputs" on page 7-465.

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 7-461. Also see "Handling and Recognizing Invalid Inputs and Outputs" on page 7-465.

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 7-461.
- 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 7-462):
    - LSFs — frequencies, w_k, where $0 < w_k < \pi$ and $w_k < w_{k+1}$
    - Normalized LSFs — $w_k / 2\pi$
    - LSPs — $\cos(w_k)$

Input and Output Dimensions, Sizes, and Frame Statuses

Valid LPC Input	LSF and LSP Outputs (Always Sample-Based)
Sample-based length-M+1 row vector, $M > 0$ $\begin{bmatrix} 1 \; a_1 \; a_2 \; \ldots \; a_m \end{bmatrix}$ *Frame-based row vectors are not valid inputs.*	Sample-based length-M row vector LSF in radians: $\begin{bmatrix} w_1 \; w_2 \; \ldots \; w_m \end{bmatrix}$ LSF normalized: $\frac{1}{2\pi} \cdot \begin{bmatrix} w_1 \; w_2 \; \ldots \; w_m \end{bmatrix}$ LSP: $\begin{bmatrix} \cos(w_1) \; \cos(w_2) \; \ldots \; \cos(w_m) \end{bmatrix}$
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, $M > 0$ $(1, a_1, a_2, \ldots, a_m)$	1-D length-M unoriented vector LSF in radians: $(w_1, w_2, \ldots, w_m)$ LSF normalized: $\frac{1}{2\pi} \cdot (w_1, w_2, \ldots, w_m)$ LSP: $(\cos(w_1), \cos(w_2), \ldots, \cos(w_m))$

Handling and Recognizing Invalid Inputs and Outputs

The block outputs invalid data 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 7-461. 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 7-465
- “Parameters for Handling Invalid Inputs and Outputs” on page 7-466

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 , Input and Output Dimensions, Sizes, and Frame Statuses, on page 7-464. However, invalid outputs do not contain all the LSP or LSF values. Instead, invalid outputs contain none or some of the LSP and LSF values and the rest of the output vector or matrix is filled with *place holder values* (-1, 0.5, or π depending on the **Output** parameter setting).

In short, all invalid outputs end in one of the place holder values (-1, 0.5, or π) as illustrated in the following table. To learn how to use the block’s parameters for handling invalid inputs and outputs, see the next section.

Output Parameter Setting	Place Holder	Sample Invalid Outputs
`LSF in radians (0 pi)`	π	$\begin{bmatrix} w_1 & w_2 & w_3 & \pi & \pi & \pi & \pi & \pi \end{bmatrix}$
`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:

- **Show output validity status (1=valid, 0=invalid)** — Setting this parameter activates a second block output port that outputs a `1` when the output is valid, and a `0` when they are invalid. The LSF and LSP outputs are *invalid* 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 7-461). See the previous section to learn how to recognize invalid outputs.
- **If current output is invalid, overwrite with previous output** — Selecting this check box causes the block to overwrite invalid outputs with the *previous* output. 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 *selecting* this check box and specifying the values in the parameter described next). The default initial overwrite values are the LSF or LSP representations of an all-pass filter.
  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 selected the **When first output is invalid, overwrite with user-defined values**. The vector of LSP/LSF values you specify should have the same dimension, size, and frame status as the other outputs, which you can look up in Table , Input and Output Dimensions, Sizes, and Frame Statuses, on page 7-464.
- **If first input value is 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 7-474.

To compute *LSP outputs*, the block relies on the fact that LSP values are the *roots of two particular polynomials* related to the input LPC polynomial; the block finds these roots using the Chebyshev polynomial root finding method, described next. To compute *LSF outputs*, the block computes the arc cosine of the LSPs, outputting values ranging from 0 to π radians.

Root Finding Method. LSPs, which are the *roots of two particular polynomials*, always lie in the range (-1, 1). (The guaranteed roots at 1 and -1 are factored out.) The block finds the LSPs by looking for a sign change of the two polynomials' values between points in the range (-1, 1). The block searches a maximum of $k(n-1)$ points, where

- n is the value of the **Root finding coarse grid points** parameter
- k is the value of the **Root finding bisection refinement** parameter

The block's method for choosing which points to check consists of the following two steps:

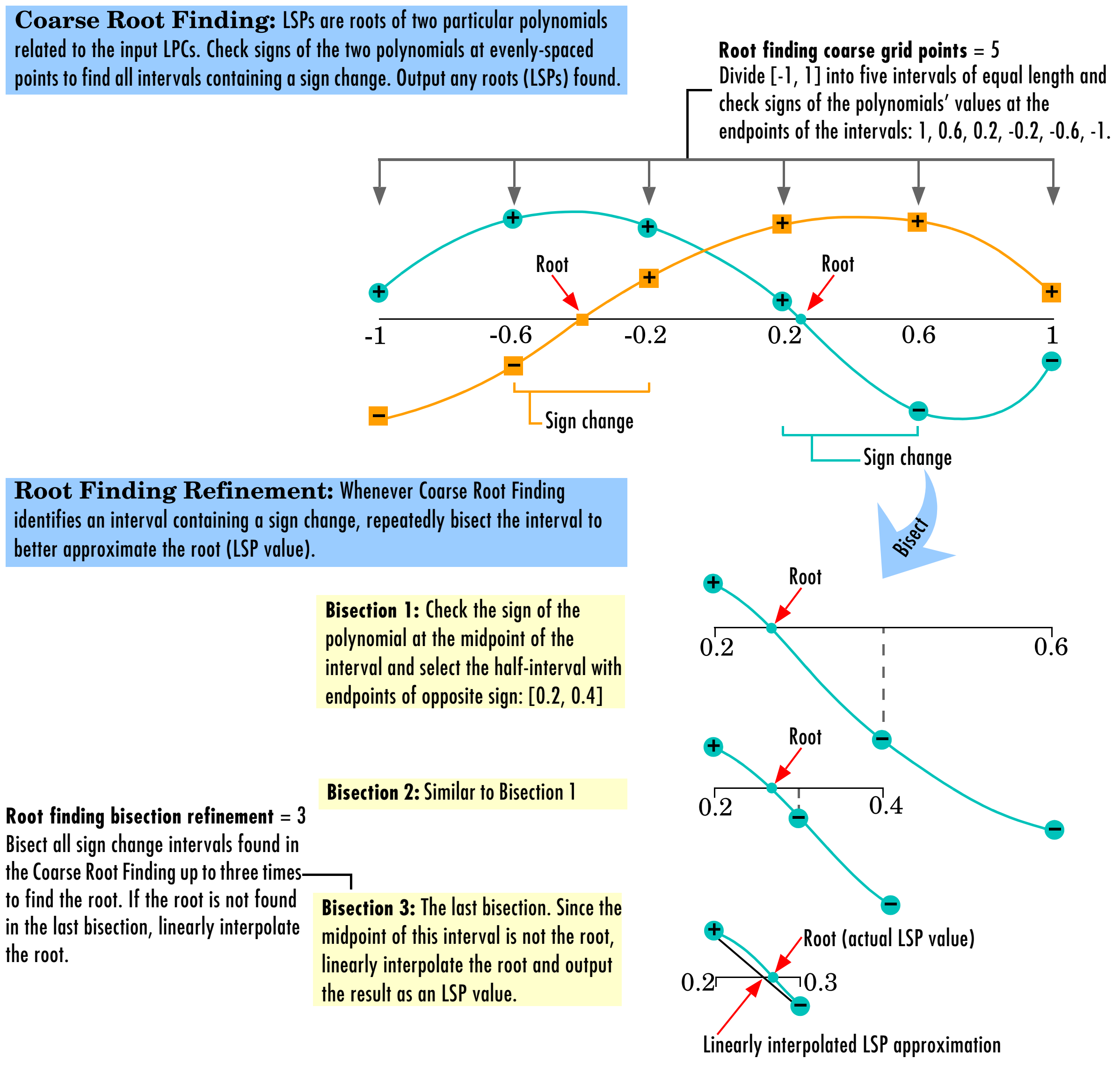
1. **Coarse Root Finding** — The block divides the interval [-1, 1] into n intervals, each of length $2/n$, and checks the signs of both polynomials' values at the endpoints of the intervals. The block starts checking signs at 1, and continues checking signs at $1-4/n$, $1-6/n$, and so on at steps of length $2/n$, outputting any point if it is a root. The block *stops searching* in these situations:
   a. The block finds a sign change of a polynomial's values between two adjacent points. An interval containing a sign change is guaranteed to contain a root, so the block further searches the interval as described in Step 2, Root Finding Refinement.
   b. The block finds and outputs all M roots (given a length-M+1 LPC input).

c The block fails to find all M roots and yields invalid outputs as described in “Handling and Recognizing Invalid Inputs and Outputs” on page 7-465.

2 **Root Finding Refinement** — When the block finds a sign change in an interval, $[a, b]$, it searches for the root guaranteed to lie in the interval by following these steps:

a **Check if Midpoint Is a Root** — The block checks the sign of the midpoint of the interval $[a, b]$. The block outputs the midpoint if it is a root, and continues Step 1, Coarse Root Finding, at the next point, $a - 2/n$. Otherwise, the block selects the half-interval with endpoints of opposite sign (either $[a, (a+b)/2]$ or $[(a+b)/2, b]$) and executes Step 2b, Stop or Continue Root Finding Refinement.

b **Stop or Continue Root Finding Refinement** — If the block has repeated Step 2a k times (k is the value of the **Root finding bisection refinement** parameter), the block linearly interpolates the root by using the half-interval’s endpoints, outputs the result as an LSP value, and returns to Step 1, Coarse Root Finding. Otherwise, the block repeats Step 2a using the half-interval.



Coarse Root Finding: LSPs are roots of two particular polynomials related to the input LPCs. Check signs of the two polynomials at evenly-spaced points to find all intervals containing a sign change. Output any roots (LSPs) found.

Root finding coarse grid points = 5
Divide [-1, 1] into five intervals of equal length and check signs of the polynomials' values at the endpoints of the intervals: 1, 0.6, 0.2, -0.2, -0.6, -1.

Root Finding Refinement: Whenever Coarse Root Finding identifies an interval containing a sign change, repeatedly bisect the interval to better approximate the root (LSP value).

Bisection 1: Check the sign of the polynomial at the midpoint of the interval and select the half-interval with endpoints of opposite sign: [0.2, 0.4]

Bisection 2: Similar to Bisection 1

Root finding bisection refinement = 3
Bisect all sign change intervals found in the Coarse Root Finding up to three times to find the root. If the root is not found in the last bisection, linearly interpolate the root.

Bisection 3: The last bisection. Since the midpoint of this interval is not the root, linearly interpolate the root and output the result as an LSP value.

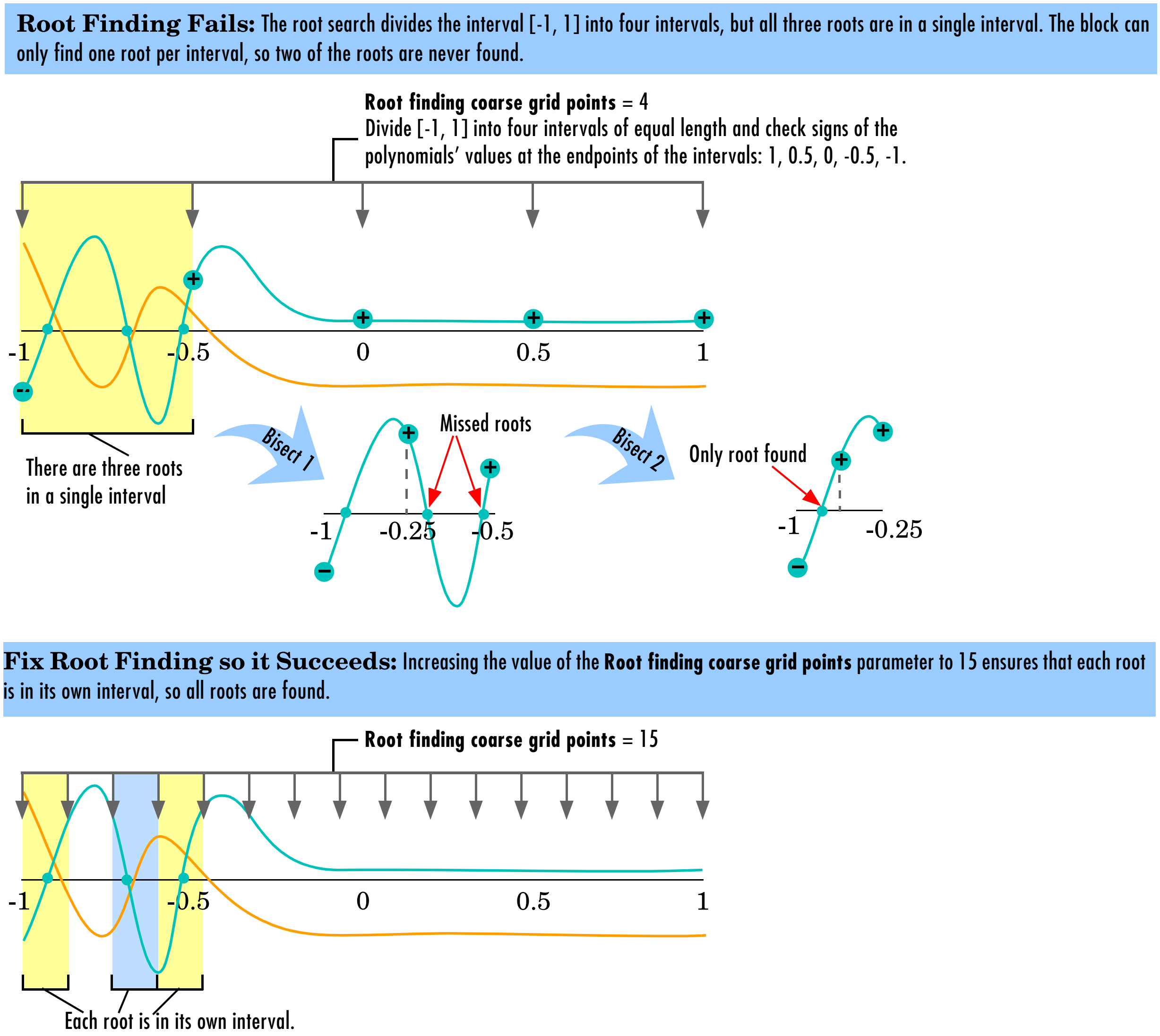
Coarse Root Finding and Root Finding Refinement

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 7-465).

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 “Fixing a Failed Root Finding” on page 7-471.

For higher-order input LPC polynomials, you should increase the **Root finding coarse grid points** value to ensure the block finds all the roots and produces valid outputs.



Root Finding Fails: The root search divides the interval [-1, 1] into four intervals, but all three roots are in a single interval. The block can only find one root per interval, so two of the roots are never found.

Root finding coarse grid points = 4
Divide [-1, 1] into four intervals of equal length and check signs of the polynomials' values at the endpoints of the intervals: 1, 0.5, 0, -0.5, -1.

There are three roots in a single interval

Missed roots

Only root found

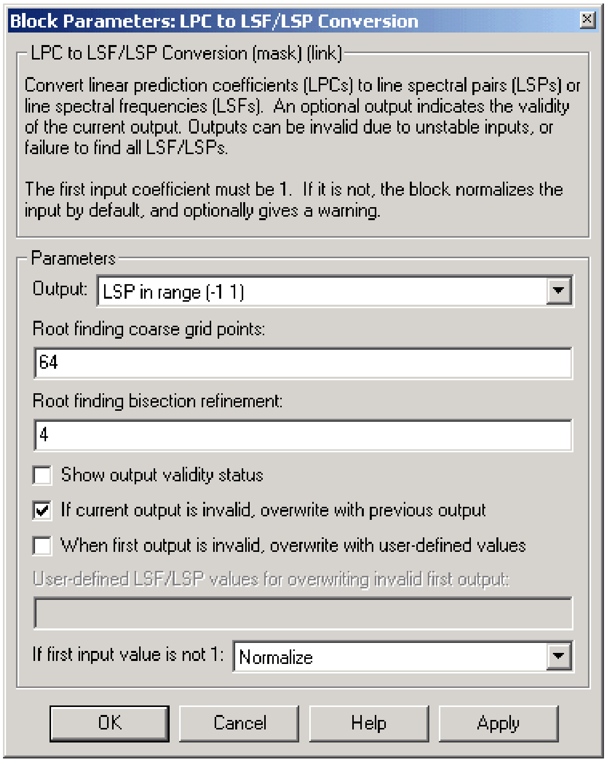
Fix Root Finding so it Succeeds: Increasing the value of the **Root finding coarse grid points** parameter to 15 ensures that each root is in its own interval, so all roots are found.

Root finding coarse grid points = 15

Each root is in its own interval.

Fixing a Failed Root Finding

Dialog Box



Block Parameters: LPC to LSF/LSP Conversion

LPC to LSF/LSP Conversion (mask) (link)

Convert linear prediction coefficients (LPCs) to line spectral pairs (LSPs) or line spectral frequencies (LSFs). An optional output indicates the validity of the current output. Outputs can be invalid due to unstable inputs, or failure to find all LSF/LSPs.

The first input coefficient must be 1. If it is not, the block normalizes the input by default, and optionally gives a warning.

Parameters

Output: LSP in range (-1 1)

Root finding coarse grid points: 64

Root finding bisection refinement: 4

□ Show output validity status

☑ If current output is invalid, overwrite with previous output

□ When first output is invalid, overwrite with user-defined values

User-defined LSF/LSP values for overwriting invalid first output:

If first input value is not 1: Normalize

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 7-462 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 7-461. 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 7-467. Also see “Adjusting Output Computation Time and Accuracy with Root Finding Parameters” on page 7-462. 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 7-467. Also see “Adjusting Output Computation Time and Accuracy with Root Finding Parameters” on page 7-462. Tunable.

Show output validity status

Selecting this check box activates a second block output port that outputs a 1 when the output is valid, and a 0 when they are invalid. For more information, see “Handling and Recognizing Invalid Inputs and Outputs” on page 7-465.

If current output is invalid, overwrite with previous output

Selecting this check box causes the block to overwrite invalid outputs with the *previous* output. Setting this parameter activates other parameters for taking care of initial overwrite values (when the very first output of the block is invalid). For more information, see “Parameters for Handling Invalid Inputs and Outputs” on page 7-466.

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 check box) or with values you specify (by *setting* this check box). The default initial overwrite values are the LSF or LSP representations of an all-pass filter. For more information, see “Parameters for Handling Invalid Inputs and Outputs” on page 7-466.

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 select **When first output is invalid, overwrite with user-defined values**. The vector or matrix of LSP/LSF values you specify should have the same dimension, size, and frame status as the other outputs, which you can look up in the table called “Input and Output Dimensions, Sizes, and Frame Statuses” on page 7-464.

If first input value is not 1

Determines what the block does when the first coefficient of an input is not `1`. The block can either proceed with computations as 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 7-466.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Boolean — Supported only by the optional output port that appears when you set the parameter, **Show output validity status (1=valid, 0=invalid)**

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

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
LPC to/from RC	DSP Blockset
LPC/RC to Autocorrelation	DSP Blockset
`poly2lsf`	Signal Processing Toolbox

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

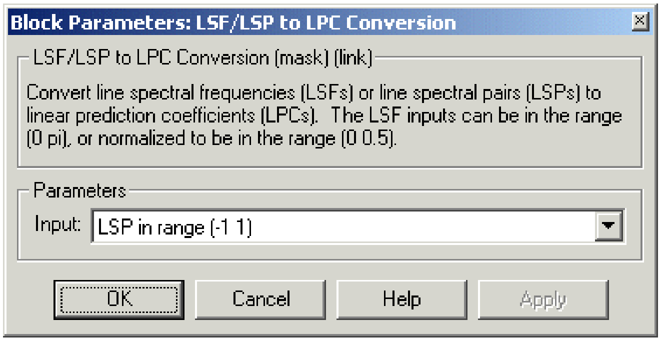
Description

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 range (0 pi)` — Vector of LSF values between 0 and π radians in increasing order. Do not include the guaranteed LSF values, 0 and π.
- `LSF normalized in range (0 0.5)` — Vector of *normalized* LSF values in increasing order, (compute by dividing the LSF values between 0 and π radians by 2π). Do not include the guaranteed normalized LSF values, 0 and 0.5.
- `LSP in range (-1 1)` — Vector of LSP values in decreasing order, equal to the cosine of the LSF values between 0 and π radians. Do not include the guaranteed LSP values, -1 and 1.

Dialog Box



Block Parameters: LSF/LSP to LPC Conversion

LSF/LSP to LPC Conversion (mask) (link)

Convert line spectral frequencies (LSFs) or line spectral pairs (LSPs) to linear prediction coefficients (LPCs). The LSF inputs can be in the range (0 pi), or normalized to be in the range (0 0.5).

Parameters

Input: LSP in range (-1 1)

Input

Specifies whether to convert `LSP in range (-1 1)`, `LSF in range (0 pi)`, or `LSF normalized in range (0 0.5)` to linear prediction coefficients (LPCs).

Supported Data Types

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

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

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
LPC to/from RC	DSP Blockset
LPC/RC to Autocorrelation	DSP Blockset
`lsf2poly`	Signal Processing Toolbox

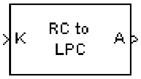
Purpose

Convert linear prediction coefficients (LPCs) to reflection coefficients (RCs) or reflection coefficients to linear prediction coefficients

Library

Estimation / Linear Prediction

Description



LPC to/from RC

The LPC to/from RC block either converts linear prediction coefficients (LPCs) to reflection coefficients (RCs) or reflection coefficients to linear prediction coefficients. Set the **Type of conversion** parameter to `LPC to RC` or `RC to LPC` to select the domain into which you want to convert your coefficients. The `A` port corresponds to LPC coefficients, and the `K` port corresponds to the RC coefficients. For more information, see “Algorithm” on page 7-478.

Consider a signal $x(n)$ as the input to an FIR analysis filter represented by LPC coefficients. The output of the this analysis filter, $e(n)$, is known as the prediction error signal. The power of this error signal is denoted by P. If the zero lag autocorrelation coefficient of $x(n)$ is one, the autocorrelation sequence and prediction error power are said to be normalized.

Select the **Output normalized prediction error power** check box to enable port `P`. The normalized prediction error power, a scalar, is output at port `P` and varies between zero and one.

Select the **Output LPC filter stability** check box to output the stability of the filter represented by the LPCs or RCs. The synthesis filter represented by the LPCs is stable if the absolute value of each of the roots of the LPC polynomial is less than one. The lattice filter represented by the RCs is stable if the absolute value of each reflection coefficient is less than 1. If the filter is stable, the block outputs a Boolean value of 1 at the `S` port. If the filter is unstable, the block outputs a Boolean value of 0 at the `S` port.

If first input value is not 1 parameter specifies the behavior of the block when the first coefficient of the LPC coefficient vector is not 1. The following options are available:

- `Replace it with 1`— Changes the first value of the coefficient vector to 1. The other coefficient values are unchanged.
- `Normalize` — Divides the entire vector of coefficients by the first coefficient so that the first coefficient of the LPC coefficient vector is 1.
- `Normalize and Warn` — Divides the entire vector of coefficients by the first coefficient so that the first coefficient of the LPC coefficient vector is 1. The

block displays a warning message telling you that your vector of coefficients has been normalized.

- `Error` — Displays an error telling you that the first coefficient of the LPC coefficient vector is not 1.

Algorithm

LPC to RC

When in this mode, this block uses backward Levinson recursion to convert linear prediction coefficients (LPCs) to reflection coefficients (RCs).For a given Nth order LPC vector$LPC_N = \left[1\ a_{N1}\ a_{N2}\ \dots\ a_{NN}\right]$, the block calculates the Nth reflection coefficient value using the formula$\gamma_N = -a_{NN}$. The block then finds the lower order LPC vectors, $LPC_{N-1}, LPC_{N-2}, \dots, LPC_1$, using the following recursion.

for $p = N, N-1, \dots, 2$,

$$\gamma_P = a_{PP}$$

$$F = 1 - \gamma_P^{\ 2}$$

$$a_{p-1,m} = \frac{a_{p,m}}{F} - \frac{\gamma_p a_{p,p-m}}{F}, 1 \le m < p$$

end

Finally, $\gamma_1 = -a_{11}$. The reflection coefficient vector is $\left[\gamma_1, \gamma_2, \dots, \gamma_N\right]$.

RC to LPC

When in this mode, this block uses Levinson recursion to convert reflection coefficients (RCs) to linear prediction coefficients (LPCs).In this case, the input to the block is $RC = \left[\gamma_1\ \gamma_2\ \dots\ \gamma_N\right]$. The zeroth order LPC vector term is 1. Starting with this term, the block uses recursion to calculate the higher order

LPC vectors,$LPC_2, LPC_3, \ldots, LPC_N$, until it has calculated the entire LPC matrix.

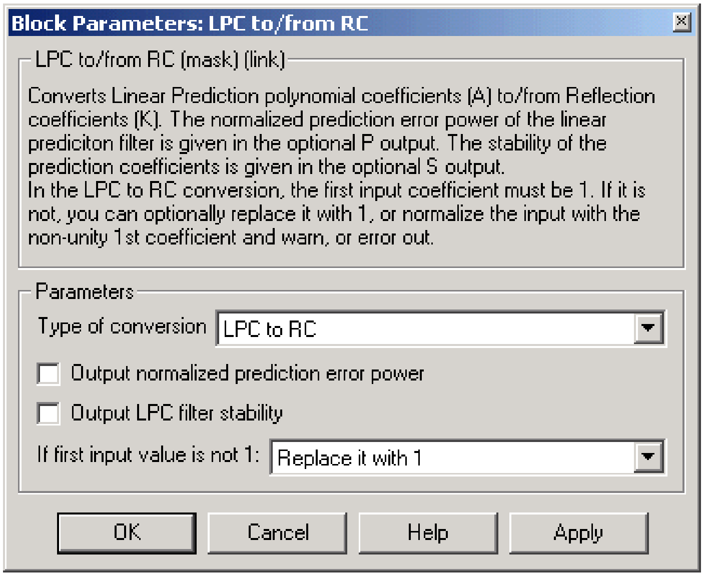
$$LPC_{matrix} = \begin{bmatrix} LPC_0 \\ LPC_1 \\ LPC_2 \\ \ldots \\ LPC_N \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & \ldots & 0 \\ 1 & a_{11} & 0 & 0 & \ldots & 0 \\ 1 & a_{21} & a_{22} & 0 & \ldots & 0 \\ 1 & a_{31} & a_{32} & a_{33} & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ 1 & a_{N1} & a_{N2} & a_{N3} & \ldots & a_{NN} \end{bmatrix}$$

This LPC matrix consists of LPC vectors of order 0 through N found by using the Levinson recursion. The following are the formulas for the recursion steps, for $p = 0, 1, \ldots, N-1$.

$$a_{p+1, m} = a_{p, m} + \gamma_{p+1} a_{p, p+1-m}, 1 \le m \le p$$

$$a_{p+1, p+1} = \gamma_{p+1}$$

Dialog Box



Block Parameters: LPC to/from RC

LPC to/from RC (mask) (link)

Converts Linear Prediction polynomial coefficients (A) to/from Reflection coefficients (K). The normalized prediction error power of the linear prediciton filter is given in the optional P output. The stability of the prediction coefficients is given in the optional S output.
In the LPC to RC conversion, the first input coefficient must be 1. If it is not, you can optionally replace it with 1, or normalize the input with the non-unity 1st coefficient and warn, or error out.

Parameters

Type of conversion: LPC to RC

□ Output normalized prediction error power

□ Output LPC filter stability

If first input value is not 1: Replace it with 1

Type of conversion

Select `LPC to RC` or `RC to LPC` to select the domain into which you want to convert your coefficients.

Output normalized prediction error power

Select this check box to output the normalized prediction error power at port `P`.

Output LPC filter stability

Select this check box to output the stability of the filter. If the filter represented by the LPCs or RCs is stable, the block outputs a Boolean value of 1 at the `S` port. If the filter represented by the LPCs or RCs is unstable, the block outputs a Boolean value of 0 at the `S` port.

If first input value is not 1

Select what you would like the block to do if the first coefficient of the LPC coefficient vector is not 1. You can choose `Replace it with 1`, `Normalize`, `Normalize and Warn`, and `Error`.

References

Makhoul, J *Linear Prediction: A tutorial review*. Proc. IEEE. 63, 63, 56 (1975).

Markel, J.D. and A. H. Gray, Jr., *Linear Prediction of Speech.* New York, Springer-Verlag, 1976.

Supported Data Types

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

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

See Also

Levinson-Durbin	DSP Blockset
LPC to LSF/LSP Conversion	DSP Blockset
LSF/LSP to LPC Conversion	DSP Blockset
LPC/RC to Autocorrelation	DSP Blockset

LPC/RC to Autocorrelation

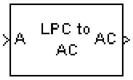
Purpose

Convert linear prediction coefficients (LPCs) or reflection coefficients (RCs) to autocorrelation coefficients (ACs)

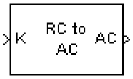
Library

Estimation / Linear Prediction

Description



LPC/RC to Autocorrelation



LPC/RC to Autocorrelation

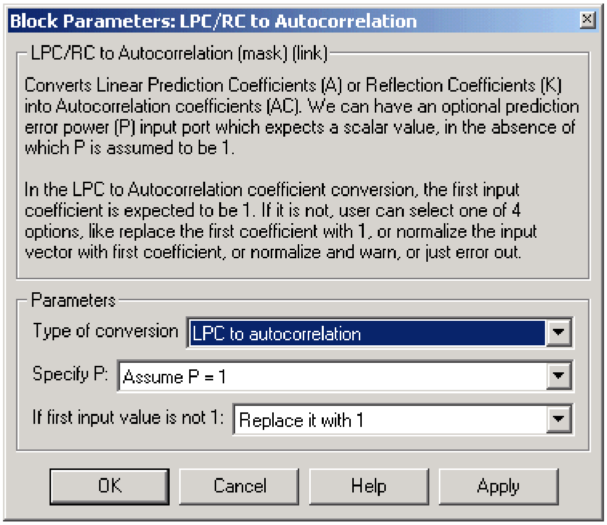
The LPC/RC to Autocorrelation block either converts linear prediction coefficients (LPCs) to autocorrelation coefficients (ACs) or reflection coefficients (RCs) to autocorrelation coefficients (ACs). Set the **Type of conversion** parameter to `LPC to autocorrelation` or `RC to autocorrelation` to select the domain from which you want to convert your coefficients. The A port corresponds to LPC coefficients, and the K port corresponds to the RC coefficients.

Use the **Specify P** parameter to set the value of the prediction error power. You can set this parameter to 1 by selecting `Assume P=1.` If you select `Via input port`, a P port appears on the block. You can use this port to input the value of the actual, non-unity prediction error power.

The **If first input value is not 1** parameter specifies the behavior of the block when the first coefficient of the LPC coefficient vector is not 1. The following options are available:

- `Replace it with 1`— The block changes the first value of the coefficient vector to 1. The rest of the coefficient values are unchanged.
- `Normalize` — The block divides the entire vector of coefficients by the first coefficient so that the first coefficient of the LPC coefficient vector is 1.
- `Normalize and Warn` — The block divides the entire vector of coefficients by the first coefficient so that the first coefficient of the LPC coefficient vector is 1. The block displays a warning message telling you that your vector of coefficients has been normalized.
- `Error` — The block displays an error telling you that the first coefficient of the LPC coefficient vector is not 1.

Dialog Box



Block Parameters: LPC/RC to Autocorrelation

LPC/RC to Autocorrelation (mask) (link)

Converts Linear Prediction Coefficients (A) or Reflection Coefficients (K) into Autocorrelation coefficients (AC). We can have an optional prediction error power (P) input port which expects a scalar value, in the absence of which P is assumed to be 1.

In the LPC to Autocorrelation coefficient conversion, the first input coefficient is expected to be 1. If it is not, user can select one of 4 options, like replace the first coefficient with 1, or normalize the input vector with first coefficient, or normalize and warn, or just error out.

Parameters

Type of conversion: LPC to autocorrelation

Specify P: Assume P = 1

If first input value is not 1: Replace it with 1

Type of conversion

From the list select `LPC to autocorrelation` or `RC to autocorrelation` to specify the domain from which you want to convert your coefficients.

Specify P

From the list select `Assume P=1` or `Via input port` to specify the value of prediction error power.

If first input value is not 1

Select what you would like the block to do if the first coefficient of the LPC coefficient vector is not 1. You can choose `Replace it with 1`, `Normalize`, `Normalize and Warn`, and `Error`.

References

Orfanidis, S.J. *Optimum Signal Processing*. New York, McGraw-Hill, 1988.

Makhoul, J. *Linear Prediction: A tutorial review*. Proc. IEEE. 63, 63, 56 (1975).

Markel, J.D. and A. H. Gray, Jr., *Linear Prediction of Speech*. New York, Springer-Verlag, 1976.

Supported Data Types

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

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

See Also

Levinson-Durbin	DSP Blockset
LPC to LSF/LSP Conversion	DSP Blockset
LSF/LSP to LPC Conversion	DSP Blockset
LPC to/from RC	DSP Blockset

LU Factorization

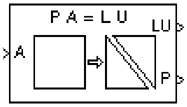
Purpose

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
```

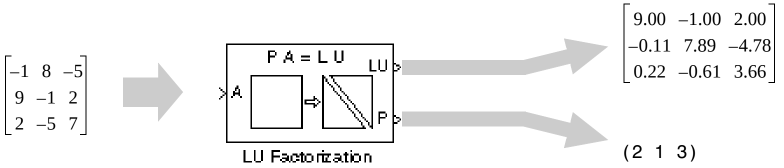
The output at the `LU` port is a composite matrix with lower subtriangle elements from L and upper triangle elements from U, and is always sample based.

Examples

The row-pivoted matrix A_p and permutation index vector P computed by the block are shown below for 3-by-3 input matrix A.

$$A = \begin{bmatrix} -1 & 8 & -5 \\ 9 & -1 & 2 \\ 2 & -5 & 7 \end{bmatrix} \qquad P = (2\ 1\ 3) \qquad A_p = \begin{bmatrix} 9 & -1 & 2 \\ -1 & 8 & -5 \\ 2 & -5 & 7 \end{bmatrix}$$

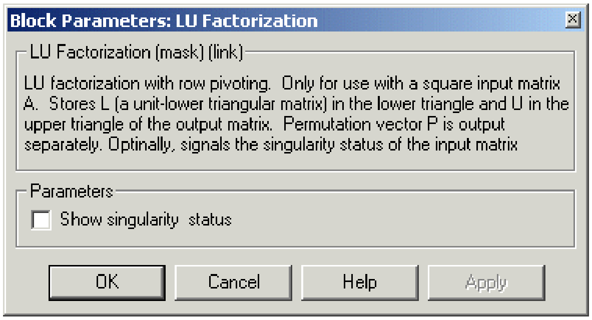
The `LU` output is a composite matrix whose lower subtriangle forms L and whose upper triangle forms U.



$$\begin{bmatrix} -1 & 8 & -5 \\ 9 & -1 & 2 \\ 2 & -5 & 7 \end{bmatrix} \qquad \begin{bmatrix} 9.00 & -1.00 & 2.00 \\ -0.11 & 7.89 & -4.78 \\ 0.22 & -0.61 & 3.66 \end{bmatrix}$$

$$L = \begin{bmatrix} 1 & 0 & 0 \\ -0.11 & 1 & 0 \\ 0.22 & -0.61 & 1 \end{bmatrix} \qquad U = \begin{bmatrix} 9.00 & -1.00 & 2.00 \\ 0 & 7.89 & -4.78 \\ 0 & 0 & 3.66 \end{bmatrix}$$

Dialog Box



Block Parameters: LU Factorization

LU Factorization (mask) (link)

LU factorization with row pivoting. Only for use with a square input matrix A. Stores L (a unit-lower triangular matrix) in the lower triangle and U in the upper triangle of the output matrix. Permutation vector P is output separately. Optinally, signals the singularity status of the input matrix

Parameters

□ Show singularity status

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 select the **Show singularity status** check box.

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

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 5-8 for related information.

LU Inverse

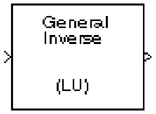
Purpose

Compute the inverse of a square matrix using LU factorization

Library

Math Functions / Matrices and Linear Algebra / Matrix Inverses

Description

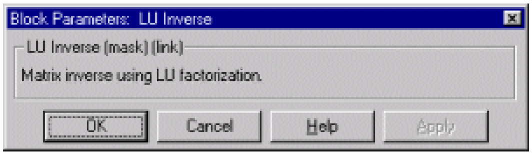


The LU Inverse block computes the inverse of the square input matrix A by factoring and inverting row-pivoted variant A_p.

$$A_p^{-1} = (LU)^{-1}$$

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



Block Parameters: LU Inverse

LU Inverse (mask) (link)

Matrix inverse using LU factorization.

References

Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

Supported Data Types

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

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

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 5-9 for related information.

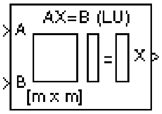
Purpose

Solve the equation A*X*=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 A*X*=B by applying LU factorization to the M-by-M matrix at the `A` port. The input to the `B` port is the right side M-by-N matrix, *B*. The output is the unique solution of the equations, M-by-N matrix *X*, and is always sample based.

A length-M 1-D vector input for right side *B* is treated as an M-by-1 matrix.

Algorithm

The LU algorithm factors a row-permuted variant (A_p) of the square input matrix A as

$$A_p = LU$$

where L is a lower-triangular square matrix with unity diagonal elements, and U is an upper-triangular square matrix.

The matrix factors are substituted for A_p in

$$A_pX = B_p$$

where B_p is the row-permuted variant of B, and the resulting equation

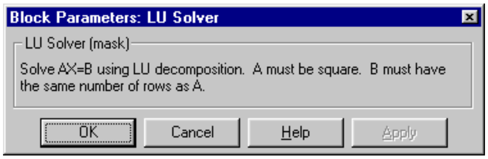
$$LUX = B_p$$

is solved for *X* by making the substitution *Y* = U*X*, and solving two triangular systems.

$$LY = B_p$$

$$UX = Y$$

Dialog Box



LU Solver (mask)

Solve AX=B using LU decomposition. A must be square. B must have the same number of rows as A.

Supported Data Types

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

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

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 5-6 for related information.

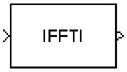
Purpose

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.

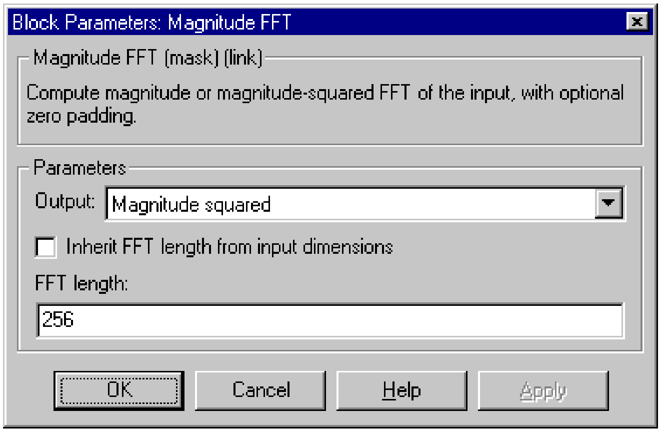
Each column of the output matrix contains the estimate of the corresponding input column's power spectral density at N_{fft} equally spaced frequency points in the range $[0,F_s)$, where F_s is the signal's sample frequency. The output is always sample based.

The block does not accept sample-based 1-by-N row vector inputs.

Examples

The `dspsacomp` demo compares the periodogram method with several other spectral estimation methods.

Dialog Box



Block Parameters: Magnitude FFT

Magnitude FFT (mask) (link)

Compute magnitude or magnitude-squared FFT of the input, with optional zero padding.

Parameters

Output: Magnitude squared

□ Inherit FFT length from input dimensions

FFT length:

256

Output

Determines whether the block computes the magnitude FFT (`Magnitude`) or magnitude-squared FFT (`Magnitude squared`) of the input. Nontunable.

Inherit FFT length from input dimensions

When selected, uses the input frame size as the number of data points, N_{fft}, on which to perform the FFT.

FFT 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 your data types to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-2.

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 5-5 for related information.

Matrix 1-Norm

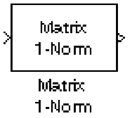
Purpose

Compute the 1-norm of a matrix.

Library

Math Functions / Matrices and Linear Algebra / Matrix Operations

Description

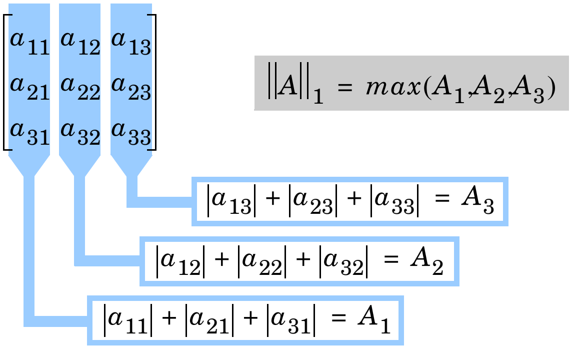


The Matrix 1-Norm block computes the 1-norm, or maximum column-sum, of an M-by-N input matrix, A.

$$y = \|A\|_1 = \max_{1 \le j \le N} \sum_{i=1}^{M} |a_{ij}|$$

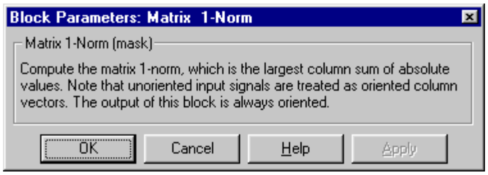
This is equivalent to

```
y = max(sum(abs(A)))    % Equivalent MATLAB code
```


$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$
$\|A\|_1 = max(A_1, A_2, A_3)$
$|a_{13}| + |a_{23}| + |a_{33}| = A_3$
$|a_{12}| + |a_{22}| + |a_{32}| = A_2$
$|a_{11}| + |a_{21}| + |a_{31}| = A_1$

A length-M 1-D vector input is treated as an M-by-1 matrix. The output, y, is always a scalar.

Dialog Box



Block Parameters: Matrix 1-Norm

Matrix 1-Norm (mask)

Compute the matrix 1-norm, which is the largest column sum of absolute values. Note that unoriented input signals are treated as oriented column vectors. The output of this block is always oriented.

References

Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

Supported Data Types

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

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

See Also

Normalization	DSP Blockset
Reciprocal Condition	DSP Blockset
`norm`	MATLAB

Matrix Exponential

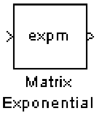
Purpose

Compute the matrix exponential

Library

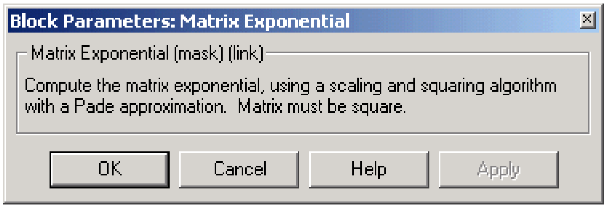
Math Functions / Matrices and Linear Algebra / Matrix Operations

Description



The Matrix Exponential block computes the matrix exponential using a scaling and squaring algorithm with a Pade approximation. The input matrix must be square.

Dialog Box



Block Parameters: Matrix Exponential

Matrix Exponential (mask) (link)

Compute the matrix exponential, using a scaling and squaring algorithm with a Pade approximation. Matrix must be square.

Supported Data Types

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

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

See Also

`expm`	MATLAB
Dot Product	Simulink
Matrix Product	DSP Blockset
Matrix Scaling	DSP Blockset
Product	Simulink

Matrix Multiply

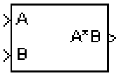
Purpose

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
```

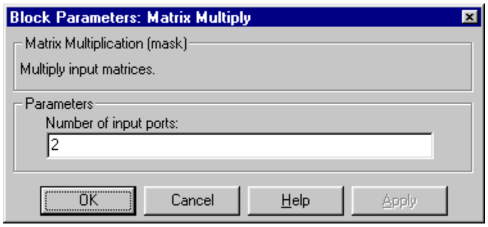
All inputs must have sizes compatible for matrix multiplication; that is, `size(A,2)` = `size(B,1)`, `size(B,2)` = `size(C,1)`, and so on. Inputs can be real, complex, sample based, or frame based in any combination, but *all* inputs must have the same precision, single or double. A length-M 1-D vector 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



Block Parameters: Matrix Multiply

Matrix Multiplication (mask)

Multiply input matrices.

Parameters

Number of input ports: 2

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 your data types to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-2.

See Also

Dot Product	Simulink
Matrix Product	DSP Blockset
Matrix Scaling	DSP Blockset
Product	Simulink

Matrix Product

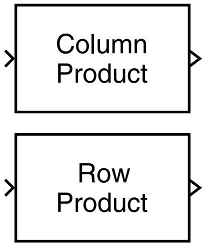
Purpose

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.

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \\ u_{31} & u_{32} & u_{33} \end{bmatrix}$$

$$\Downarrow$$

$$\begin{bmatrix} y_1 & y_2 & y_3 \end{bmatrix} = \begin{bmatrix} (u_{11}u_{21}u_{31}) & (u_{12}u_{22}u_{32}) & (u_{13}u_{23}u_{33}) \end{bmatrix}$$

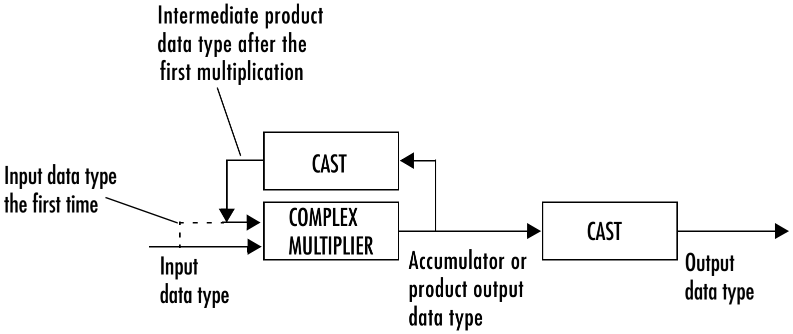
This is equivalent to

```
y = prod(u)        % Equivalent MATLAB code
```

The output of the Matrix Product block has the same frame status as the input. This block accepts real and complex floating-point and fixed-point inputs.

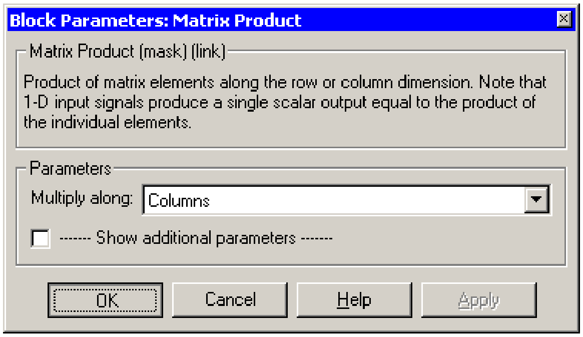
Fixed-Point Data Types

The following diagram shows the data types used within the Matrix Product block for fixed-point signals.


Intermediate product data type after the first multiplication
Input data type the first time
CAST
COMPLEX MULTIPLIER
Input data type
Accumulator or product output data type
CAST
Output data type

The output of the multiplier is in the product output data type if at least one of the inputs to the multiplier is real. If both of the inputs to the multiplier are complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed, refer to “Multiplication Data Types” on page 6-15. You can set the accumulator, product output, intermediate product, and output data types in the block mask as discussed in “Dialog Box” below.

Dialog Box



Block Parameters: Matrix Product

Matrix Product (mask) (link)

Product of matrix elements along the row or column dimension. Note that 1-D input signals produce a single scalar output equal to the product of the individual elements.

Parameters

Multiply along: Columns

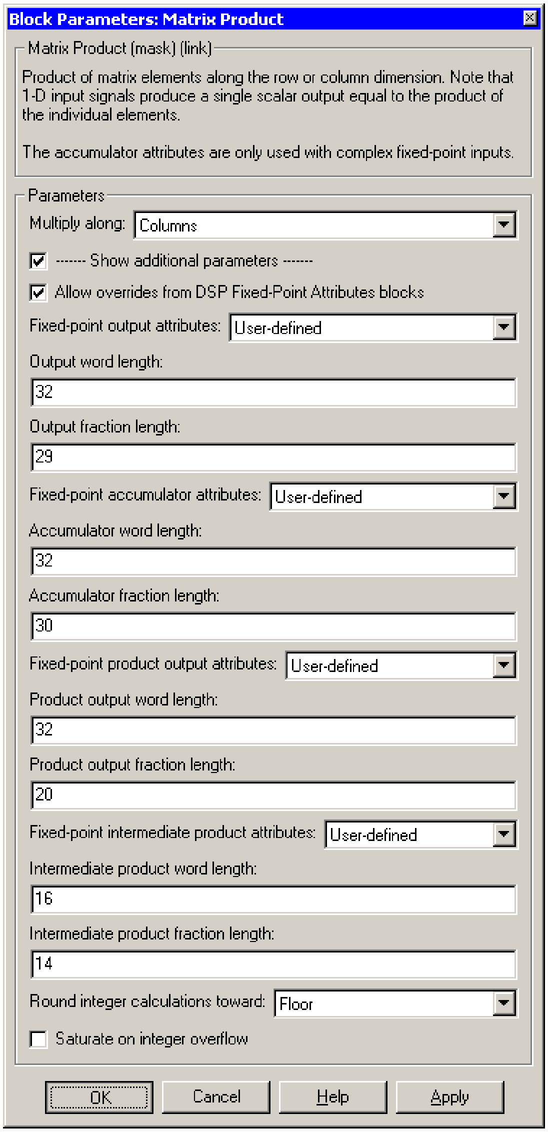
□ ------- Show additional parameters -------

Multiply along

Indicate whether to multiply together the elements of each row or of each column of the input.

Show additional parameters

If selected, additional parameters specific to implementation of the block become visible as shown.



Block Parameters: Matrix Product

Matrix Product (mask) (link)

Product of matrix elements along the row or column dimension. Note that 1-D input signals produce a single scalar output equal to the product of the individual elements.

The accumulator attributes are only used with complex fixed-point inputs.

Parameters

Multiply along: Columns

☑ ------ Show additional parameters ------

☑ Allow overrides from DSP Fixed-Point Attributes blocks

Fixed-point output attributes: User-defined

Output word length: 32

Output fraction length: 29

Fixed-point accumulator attributes: User-defined

Accumulator word length: 32

Accumulator fraction length: 30

Fixed-point product output attributes: User-defined

Product output word length: 32

Product output fraction length: 20

Fixed-point intermediate product attributes: User-defined

Intermediate product word length: 16

Intermediate product fraction length: 14

Round integer calculations toward: Floor

☐ Saturate on integer overflow

Allow overrides from DSP Fixed-Point Attributes blocks

If you select this parameter, fixed-point data types for this block may be set by DSP Fixed-Point Attributes blocks in your model. If this parameter is unselected, the data types are always set by the parameters in the block mask.

Fixed-point output attributes

Choose how you will specify the word length and fraction length of the output of the block. If you select `Same as input`, these characteristics will match those of the input to the block. If you select `User-defined`, the **Output word length** and **Output fraction length** parameters become visible.

Output word length

Specify the word length, in bits, of the output. This parameter is only visible if `User-defined` is specified for the **Fixed-point output attributes** parameter.

Output fraction length

Specify the fraction length, in bits, of the output. This parameter is only visible if `User-defined` is specified for the **Fixed-point output attributes** parameter.

Fixed-point accumulator attributes

Use this parameter to specify how you would like to designate the accumulator word and fraction lengths. Refer to “Fixed-Point Data Types” on page 7-500 and “Multiplication Data Types” on page 6-15 for illustrations depicting the use of the accumulator data type in this block. Note that the accumulator data type is only used if both inputs to the multiplier are complex.

If you select `Same as input`, the accumulator word and fraction lengths are the same as those of the input to the block. If you select `Same as output`, they are the same as those of the output of the block. If you select `User-defined`, the **Accumulator word length** and **Accumulator fraction length** parameters become visible.

Accumulator word length

Specify the word length, in bits, of the accumulator. This parameter is only visible when `User-defined` is specified for the **Fixed-point accumulator attributes** parameter.

Accumulator fraction length

Specify the fraction length, in bits, of the accumulator. This parameter is only visible when `User-defined` is specified for the **Fixed-point accumulator attributes** parameter.

Fixed-point product output attributes

Use this parameter to specify how you would like to designate the product output word and fraction lengths. Refer to “Fixed-Point Data Types” on page 7-500 and “Multiplication Data Types” on page 6-15 for illustrations depicting the use of the product output data type in this block.

If you select `Same as input`, `Same as output`, or `Same as accumulator`, the product output word and fraction lengths are the same as those of the input, output, or accumulator of the block, respectively. If you select `User-defined`, the **Product output word length** and **Product output fraction length** parameters become visible.

Product output word length

Specify the word length, in bits, of the product output. This parameter is only visible when `User-defined` is specified for the **Fixed-point product output attributes** parameter.

Product output fraction length

Specify the fraction length, in bits, of the product output. This parameter is only visible when `User-defined` is specified for the **Fixed-point product output attributes** parameter.

Fixed-point intermediate product attributes

As shown above, the output of the multiplier is cast from the product output data type to the intermediate product data type before the next element of the input is multiplied into it. Use this parameter to specify how you would like to designate the intermediate product word and fraction lengths.

If you select `Same as input`, the intermediate product word and fraction lengths are the same as those of the input. If you select `Same as output`, they are the same as those of the output. If you select `User-defined`, the **Intermediate product word length** and **Intermediate product fraction length** paramters become visible.

Intermediate product word length

Specify the word length, in bits, of the intermediate product. This parameter is only visible when `User-defined` is selected for the **Fixed-point intermediate product attributes** parameter.

Intermediate product fraction length

Specify the fraction length, in bits, of the intermediate product. This parameter is only visible when `User-defined` is selected for the **Fixed-point intermediate product attributes** parameter.

Round integer calculations towards

Select the rounding mode for fixed-point operations.

Saturate on integer overflow

If selected, overflows saturate.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point
- 8-, 16-, and 32-bit signed integers

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

See Also

Matrix Multiply	DSP Blockset
Matrix Square	DSP Blockset
Matrix Sum	DSP Blockset
`prod`	MATLAB

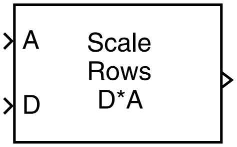
Purpose

Scale the rows or columns of a matrix by a specified vector

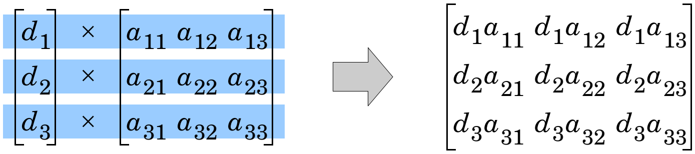
Library

Math Functions / Matrices and Linear Algebra / Matrix Operations

Description



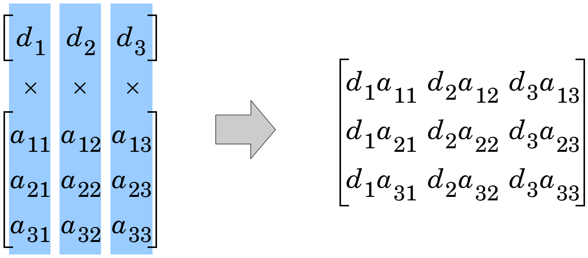
The Matrix Scaling block scales the rows or columns of the M-by-N input matrix A by the values in input vector D. When the **Mode** parameter is set to `Scale Rows (D*A)`, the input D can be a 1-D or 2-D vector of length M, and the block multiplies each element of D across the corresponding *row* of matrix A.


$\begin{bmatrix} d_1 \\ d_2 \\ d_3 \end{bmatrix} \begin{matrix} \times \\ \times \\ \times \end{matrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$
$\begin{bmatrix} d_1a_{11} & d_1a_{12} & d_1a_{13} \\ d_2a_{21} & d_2a_{22} & d_2a_{23} \\ d_3a_{31} & d_3a_{32} & d_3a_{33} \end{bmatrix}$

This is equivalent to premultiplying A by a diagonal matrix with diagonal D.

```
y = diag(D)*A      % Equivalent MATLAB code
```

When the **Mode** parameter is set to `Scale Columns (A*D)`, the input D can be a 1-D or 2-D vector of length N, and the block multiplies each element of D across the corresponding *column* of matrix A.


$\begin{bmatrix} d_1 & d_2 & d_3 \end{bmatrix}$
$\times \quad \times \quad \times$
$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$
$\begin{bmatrix} d_1a_{11} & d_2a_{12} & d_3a_{13} \\ d_1a_{21} & d_2a_{22} & d_3a_{23} \\ d_1a_{31} & d_2a_{32} & d_3a_{33} \end{bmatrix}$

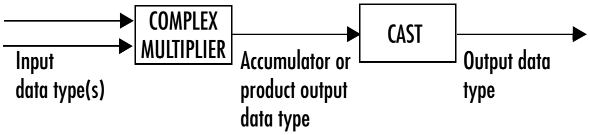
This is equivalent to postmultiplying A by a diagonal matrix with diagonal D.

```
y = A*diag(D)      % Equivalent MATLAB code
```

The output of the Matrix Scaling block 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. This block accepts real and complex floating-point and fixed-point inputs. This block does not support complex code generation.

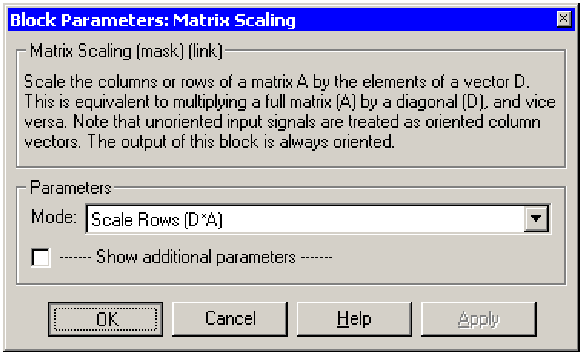
Fixed-Point Data Types

The following diagram shows the data types used within the Matrix Scaling block for fixed-point signals.



The output of the multiplier is in the product output data type if at least one of the inputs to the multiplier is real. If both of the inputs to the multiplier are complex, the result of the multiplication is in the accumulator data type. For details on the complex multiplication performed, refer to “Multiplication Data Types” on page 6-15. You can set the accumulator, product output, and output data types in the block mask as discussed below.

Dialog Box



Block Parameters: Matrix Scaling

Matrix Scaling (mask) (link)

Scale the columns or rows of a matrix A by the elements of a vector D. This is equivalent to multiplying a full matrix (A) by a diagonal (D), and vice versa. Note that unoriented input signals are treated as oriented column vectors. The output of this block is always oriented.

Parameters

Mode: Scale Rows (D*A)

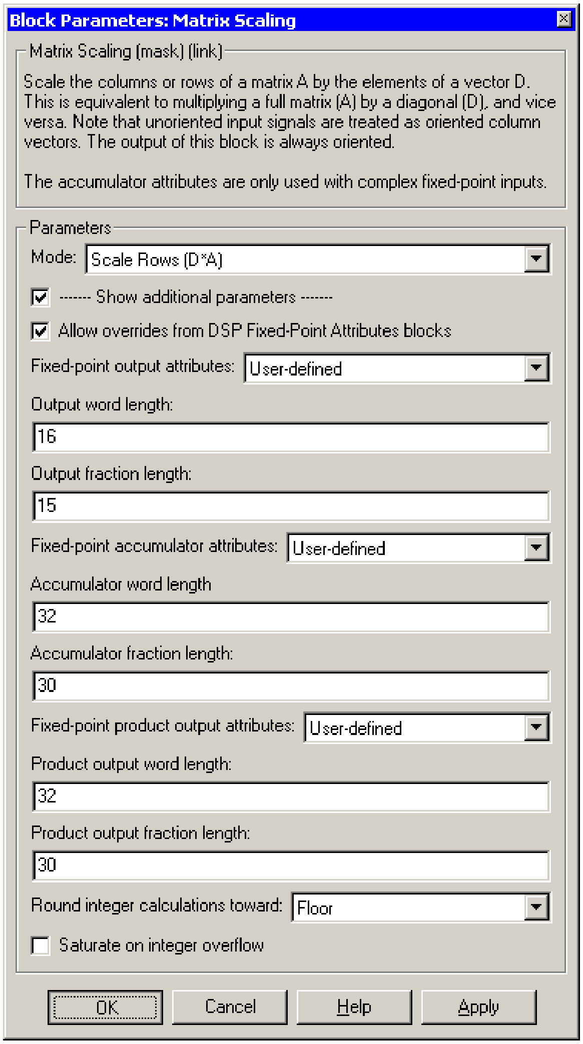
□ ------- Show additional parameters -------

Mode

Specify the mode of operation, row scaling or column scaling. Nontunable.

Show additional parameters

If selected, additional parameters specific to implementation of the block become visible as shown.



Block Parameters: Matrix Scaling

Matrix Scaling (mask) (link)

Scale the columns or rows of a matrix A by the elements of a vector D. This is equivalent to multiplying a full matrix (A) by a diagonal (D), and vice versa. Note that unoriented input signals are treated as oriented column vectors. The output of this block is always oriented.

The accumulator attributes are only used with complex fixed-point inputs.

Parameters

Mode: Scale Rows (D*A)

☑ ------ Show additional parameters ------

☑ Allow overrides from DSP Fixed-Point Attributes blocks

Fixed-point output attributes: User-defined

Output word length: 16

Output fraction length: 15

Fixed-point accumulator attributes: User-defined

Accumulator word length 32

Accumulator fraction length: 30

Fixed-point product output attributes: User-defined

Product output word length: 32

Product output fraction length: 30

Round integer calculations toward: Floor

☐ Saturate on integer overflow

Allow overrides from DSP Fixed-Point Attributes blocks

If you select this parameter, fixed-point data types for this block may be set by DSP Fixed-Point Attributes blocks in your model. If this parameter is unselected, the data types are always set by the parameters in the block mask.

Fixed-point output attributes

Choose how you will specify the output word length and fraction length. If you select `Same as first input`, these characteristics will match those of the matrix input `A`. If you select `User-defined`, the **Output word length** and **Output fraction length** parameters become visible.

Output word length

Specify the word length, in bits, of the output. This parameter is only visible if `User-defined` is specified for the **Fixed-point output attributes** parameter.

Output fraction length

Specify the fraction length, in bits, of the output. This parameter is only visible if `User-defined` is specified for the **Fixed-point output attributes** parameter.

Fixed-point accumulator attributes

Use this parameter to specify how you would like to designate the accumulator word and fraction lengths. Refer to “Fixed-Point Data Types” on page 7-506 and “Multiplication Data Types” on page 6-15 for illustrations depicting the use of the accumulator data type in this block. Note that the accumulator data type is only used if both inputs to the multiplier are complex.

If you select `Same as first input`, the accumulator word and fraction lengths are the same as those of the first input to the block. If you select `Same as output`, they are the same as those of the output of the block. If you select `User-defined`, the **Accumulator word length** and **Accumulator fraction length** parameters become visible.

Accumulator word length

Specify the word length, in bits, of the accumulator. This parameter is only visible when `User-defined` is specified for the **Fixed-point accumulator attributes** parameter.

Accumulator fraction length

Specify the fraction length, in bits, of the accumulator. This parameter is only visible when `User-defined` is specified for the **Fixed-point accumulator attributes** parameter.

Fixed-point product output attributes

Use this parameter to specify how you would like to designate the product output word and fraction lengths. Refer to “Fixed-Point Data Types” on page 7-506 and “Multiplication Data Types” on page 6-15 for illustrations depicting the use of the product output data type in this block.

If you select `Same as first input`, `Same as output`, or `Same as accumulator`, the product output word and fraction lengths are the same as those of the first input, output, or accumulator of the block, respectively. If you select `User-defined`, the **Product output word length** and **Product output fraction length** parameters become visible.

Product output word length

Specify the word length, in bits, of the product output. This parameter is only visible when `User-defined` is specified for the **Fixed-point product output attributes** parameter.

Product output fraction length

Specify the fraction length, in bits, of the product output. This parameter is only visible when `User-defined` is specified for the **Fixed-point product output attributes** parameter.

Round integer calculations toward

Select the rounding mode for fixed-point operations.

Saturate on integer overflow

If selected, overflows saturate.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point
- 8-, 16-, and 32-bit signed integers

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

See Also

Matrix Multiply	DSP Blockset
Matrix Product	DSP Blockset
Matrix Sum	DSP Blockset

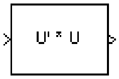
Purpose

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
```

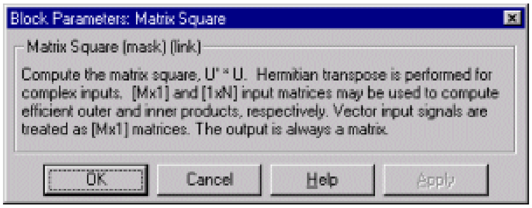
A length-M 1-D vector input is treated as an M-by-1 matrix. For both sample-based and frame-based inputs, output y is sample based with dimension N-by-N.

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



Compute the matrix square, U' * U. Hermitian transpose is performed for complex inputs. [Mx1] and [1xN] input matrices may be used to compute efficient outer and inner products, respectively. Vector input signals are treated as [Mx1] matrices. The output is always a matrix.

Supported Data Types

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

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

See Also

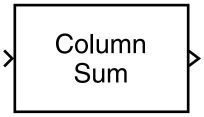
Matrix Multiply	DSP Blockset
Matrix Product	DSP Blockset
Matrix Sum	DSP Blockset
Transpose	DSP Blockset

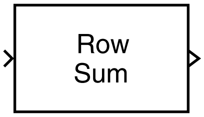
Matrix Sum

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.

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \\ u_{31} & u_{32} & u_{33} \end{bmatrix}$$

$$\Downarrow$$

$$\begin{bmatrix} y_1 & y_2 & y_3 \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{3} u_{i1} & \sum_{i=1}^{3} u_{i2} & \sum_{i=1}^{3} u_{i3} \end{bmatrix}$$

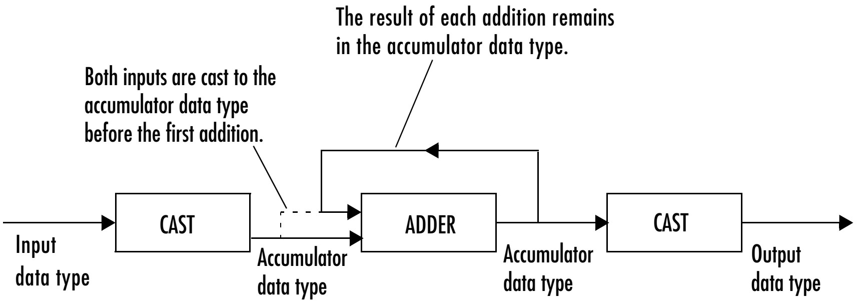
This is equivalent to

```
y = sum(u)          % Equivalent MATLAB code
```

The output of the Matrix Sum block has the same frame status as the input. This block accepts real and complex floating-point and fixed-point inputs.

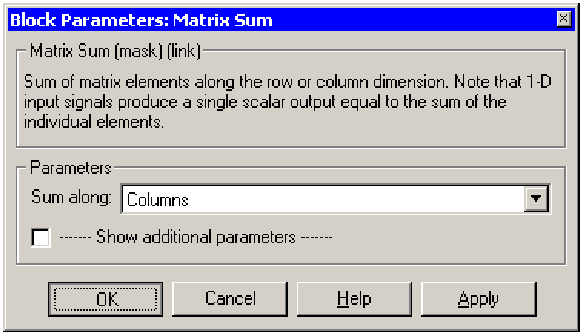
Fixed-Point Data Types

The following diagram shows the data types used within the Matrix Sum block for fixed-point signals.


The result of each addition remains in the accumulator data type.
Both inputs are cast to the accumulator data type before the first addition.
CAST
ADDER
CAST
Input data type
Accumulator data type
Accumulator data type
Output data type

You can set the accumulator and output data types in the block mask as discussed in "Dialog Box" below.

Dialog Box



Matrix Sum (mask) (link)

Sum of matrix elements along the row or column dimension. Note that 1-D input signals produce a single scalar output equal to the sum of the individual elements.

Parameters

Sum along: Columns

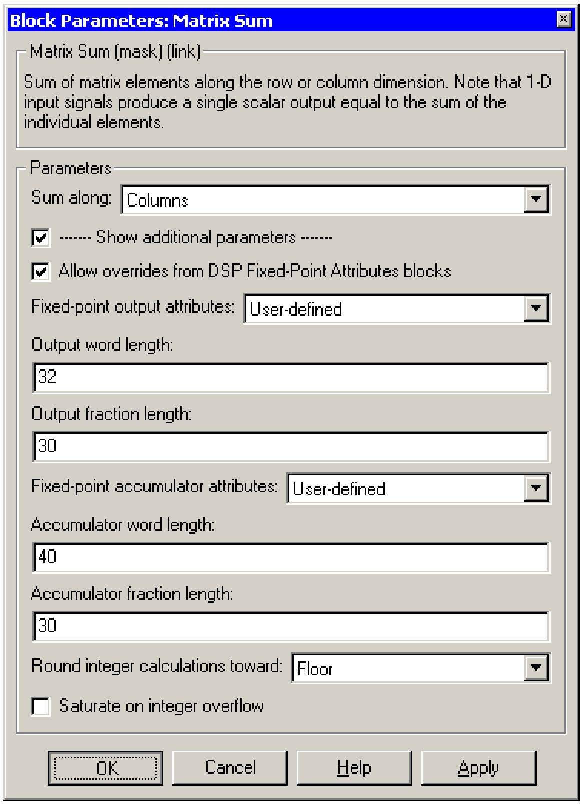
□ ------ Show additional parameters ------

Sum along

Indicate whether to sum the elements of each row or of each column of the input.

Show additional parameters

If selected, additional parameters specific to implementation of the block become visible as shown.



Block Parameters: Matrix Sum

Matrix Sum (mask) (link)

Sum of matrix elements along the row or column dimension. Note that 1-D input signals produce a single scalar output equal to the sum of the individual elements.

Parameters

Sum along: Columns

☑ ------- Show additional parameters -------

☑ Allow overrides from DSP Fixed-Point Attributes blocks

Fixed-point output attributes: User-defined

Output word length: 32

Output fraction length: 30

Fixed-point accumulator attributes: User-defined

Accumulator word length: 40

Accumulator fraction length: 30

Round integer calculations toward: Floor

☐ Saturate on integer overflow

Allow overrides from DSP Fixed-Point Attributes blocks

If you select this parameter, fixed-point data types for this block may be set by DSP Fixed-Point Attributes blocks in your model. If this parameter is unselected, the data types are always set by the parameters in the block mask.

Fixed-point output attributes

Choose how you will specify the output word length and fraction length. If you select `Same as input`, these characteristics will match those of the input to the block. If you select `User-defined`, the **Output word length** and **Output fraction length** parameters become visible.

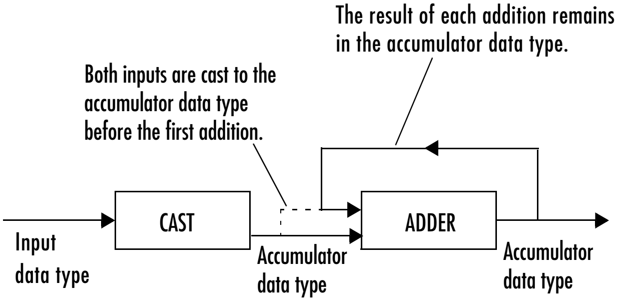
Output word length

Specify the word length, in bits, of the output. This parameter is only visible if `User-defined` is specified for the **Fixed-point output attributes** parameter.

Output fraction length

Specify the fraction length, in bits, of the output. This parameter is only visible if `User-defined` is specified for the **Fixed-point output attributes** parameter.

Fixed-point accumulator attributes


The result of each addition remains in the accumulator data type.
Both inputs are cast to the accumulator data type before the first addition.
CAST
ADDER
Input data type
Accumulator data type
Accumulator data type

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

If you select `Same as output`, the accumulator word and fraction lengths are the same as those of the output. If you select `User-defined`, the **Accumulator word length** and **Accumulator fraction length** parameters become visible.

Accumulator word length

Specify the word length, in bits, of the accumulator. This parameter is only visible when `User-defined` is specified for the **Fixed-point accumulator attributes** parameter.

Accumulator fraction length

Specify the fraction length, in bits, of the accumulator. This parameter is only visible when `User-defined` is specified for the **Fixed-point accumulator attributes** parameter.

Round integer calculations toward

Select the rounding mode for fixed-point operations.

Saturate on integer overflow

If selected, overflows saturate.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point
- 8-, 16-, and 32-bit signed integers

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

See Also

Matrix Product	DSP Blockset
Matrix Multiply	DSP Blockset
`sum`	MATLAB

Matrix Viewer

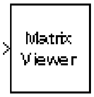
Purpose

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

Select 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 the `ColorSpec` property in the MATLAB documentation for complete information about defining RGB triples.

MATLAB provides a number of functions for generating predefined colormaps, such as `hot`, `cool`, `bone`, and `autumn`. Each of these functions accepts the colormap size as an argument, and can be used in the **Colormap matrix** parameter. For example, 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

Select 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 pop-up menu that is displayed when you right-click in the figure window.

When **Axis zoom** is cleared, the axis labels and titles are displayed in a gray border surrounding the image axes, 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:

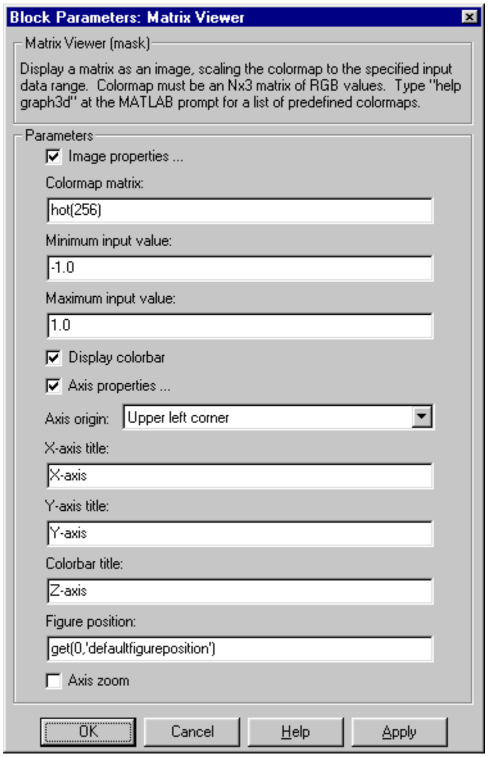
- **Refresh** erases all data on the scope display, except for the most recent image.
- **Autoscale** recomputes the **Minimum input** and **Maximum input** parameter values to best fit the range of values observed in a series of 10 consecutive inputs. The numerical limits selected by the autoscale feature 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 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



Block Parameters: Matrix Viewer

Matrix Viewer (mask)

Display a matrix as an image, scaling the colormap to the specified input data range. Colormap must be an Nx3 matrix of RGB values. Type "help graph3d" at the MATLAB prompt for a list of predefined colormaps.

Parameters

☑ Image properties ...

Colormap matrix: hot(256)

Minimum input value: -1.0

Maximum input value: 1.0

☑ Display colorbar

☑ Axis properties ...

Axis origin: Upper left corner

X-axis title: X-axis

Y-axis title: Y-axis

Colorbar title: Z-axis

Figure position: get(0,'defaultfigureposition')

□ Axis zoom

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. Right-click in the figure window and select **Autoscale** from pop-up menu to set this parameter to the minimum value observed in a series of 10 consecutive matrix inputs. Tunable.

Maximum input value

The input value to be mapped to the color defined in the last row of the colormap matrix. Right-click in the figure window and select **Autoscale** from the pop-up menu to set this parameter to the maximum value observed in a series of 10 consecutive matrix inputs. Tunable.

Display colorbar

Select to display a bar with the selected colormap to the right of the image axes. Tunable.

Axis 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 your data types to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-2.

See Also

Spectrum Scope	DSP Blockset
Vector Scope	DSP Blockset
`colormap`	MATLAB
`ColorSpec`	MATLAB
`image`	MATLAB

Also see “Viewing Signals” on page 2-87 for how to use this and other blocks to view signals.

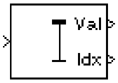
Purpose

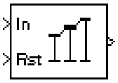
Find the maximum values in an input or sequence of inputs

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`.

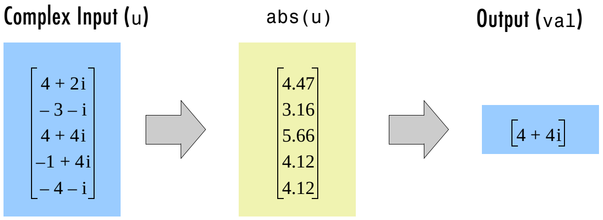
Value Mode

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.

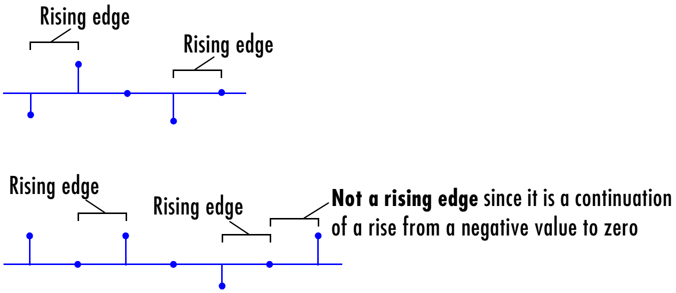
As in the other modes, length-M 1-D vector inputs and *sample-based* length-M row vector inputs are both treated as M-by-1 column vectors.

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.

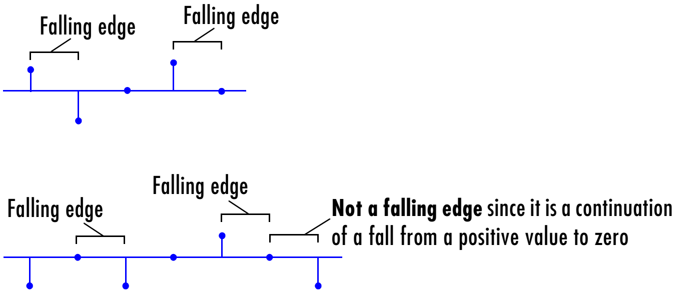
For sample-based inputs, a reset event causes the running maximum for each channel to be initialized to the value in the corresponding channel of the current input. For frame-based inputs, a reset event causes the running maximum for each channel to be initialized to the earliest value in each channel 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)


Rising edge
Rising edge
Rising edge
Rising edge
Not a rising edge since it is a continuation of a rise from a negative value to zero

- 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)

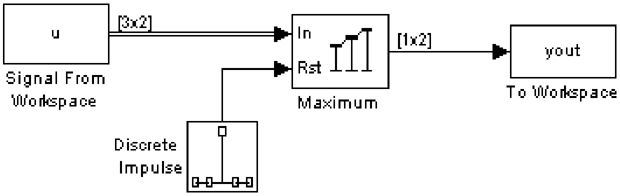

Falling edge
Falling edge
Falling edge
Falling edge
Not a falling edge since it is a continuation of a fall from a positive value to zero

- Either edge — Triggers a reset operation when the Rst input is a Rising edge or Falling edge (as described above).
- Non-zero sample — Triggers a reset operation at each sample time that the Rst input is not zero.

Note When running simulations in the Simulink **MultiTasking** mode, sample-based reset signals have a one-sample latency, and frame-based reset signals have one frame of latency. Thus, there is a one-sample or one-frame delay between the time the block detects a reset event, and when it applies the reset. For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" on page 2-98 and the topic on The Simulation Parameters Dialog Box in the Simulink documentation.

Examples

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

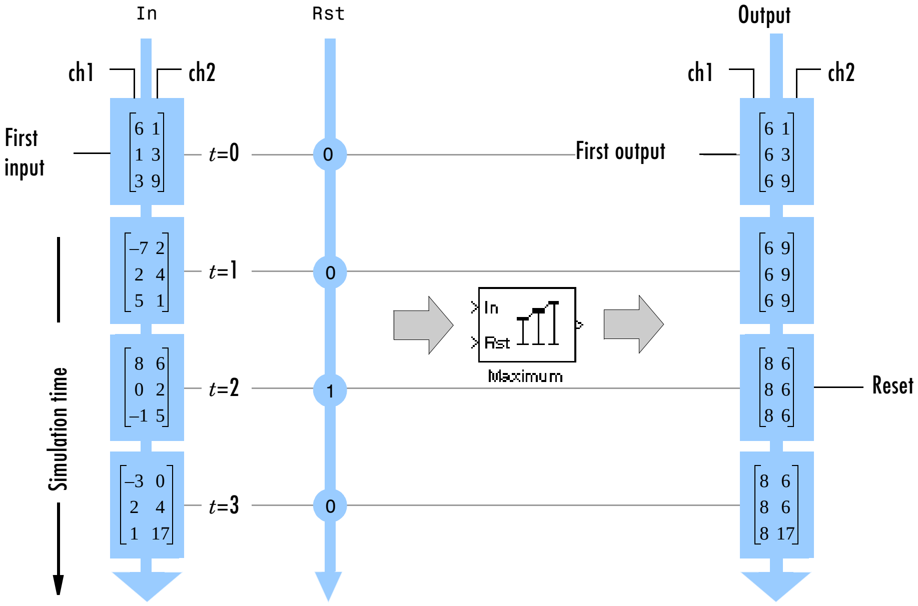
```
u = [6 1 3 -7 2 5 8 0 -1 -3 2 1;1 3 9 2 4 1 6 2 5 0 4 17]'
```

The Discrete Impulse block has the following settings:

- **Delay (samples)** = 2
- **Sample time** = 1

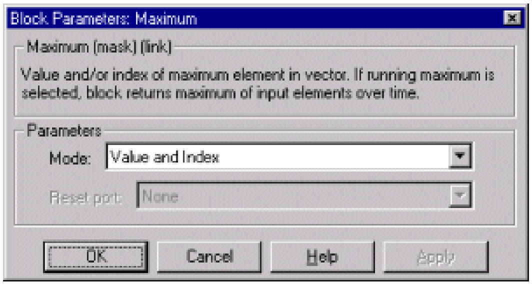
- **Samples per frame** = 1

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


Block Parameters: Maximum

Maximum (mask) (link)

Value and/or index of maximum element in vector. If running maximum is selected, block returns maximum of input elements over time.

Parameters

Mode: Value and Index

Reset port: None

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 7-525.

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 your data types to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-2.

See Also

Mean	DSP Blockset
Minimum	DSP Blockset
MinMax	Simulink
`max`	MATLAB

Mean

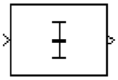
Purpose

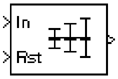
Find the mean value of an input or sequence of inputs

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.

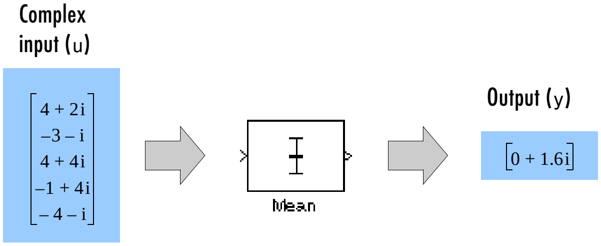
Basic Operation

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.

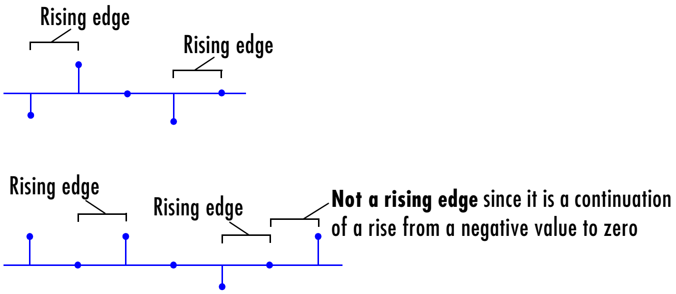
As in basic operation, length-M 1-D vector inputs and *sample-based* length-M row vector inputs are both treated as M-by-1 column vectors.

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.

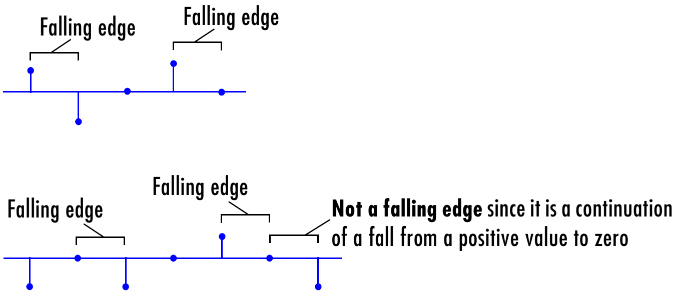
The reset event is specified by the **Reset port** parameter, and can be one of the following:

- `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)


Rising edge
Rising edge
Rising edge
Rising edge
Not a rising edge since it is a continuation of a rise from a negative value to zero

- `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)

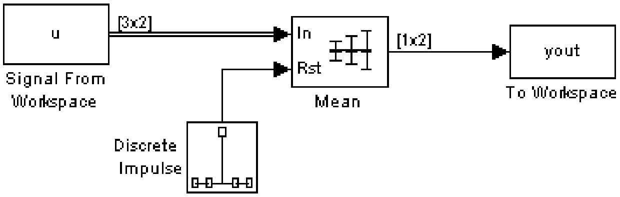

Falling edge
Falling edge
Falling edge
Falling edge
Not a falling edge since it is a continuation of a fall from a positive value to zero

- `Either edge` — Triggers a reset operation when the `Rst` input is a `Rising edge` or `Falling edge` (as described above).
- `Non-zero sample` — Triggers a reset operation at each sample time that the `Rst` input is not zero.

Note When running simulations in the Simulink `MultiTasking` mode, sample-based reset signals have a one-sample latency, and frame-based reset signals have one frame of latency. Thus, there is a one-sample or one-frame delay between the time the block detects a reset event, and when it applies the reset. For more information on latency and the Simulink tasking modes, see “Excess Algorithmic Delay (Tasking Latency)” on page 2-98 and the topic on The Simulation Parameters Dialog Box in the Simulink documentation.

Examples

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** = Select this check box
- **Reset port** = `Non-zero sample`

The Signal From Workspace block has the following settings:

- **Signal** = `u`
- **Sample time** = `1/3`
- **Samples per frame** = `3`

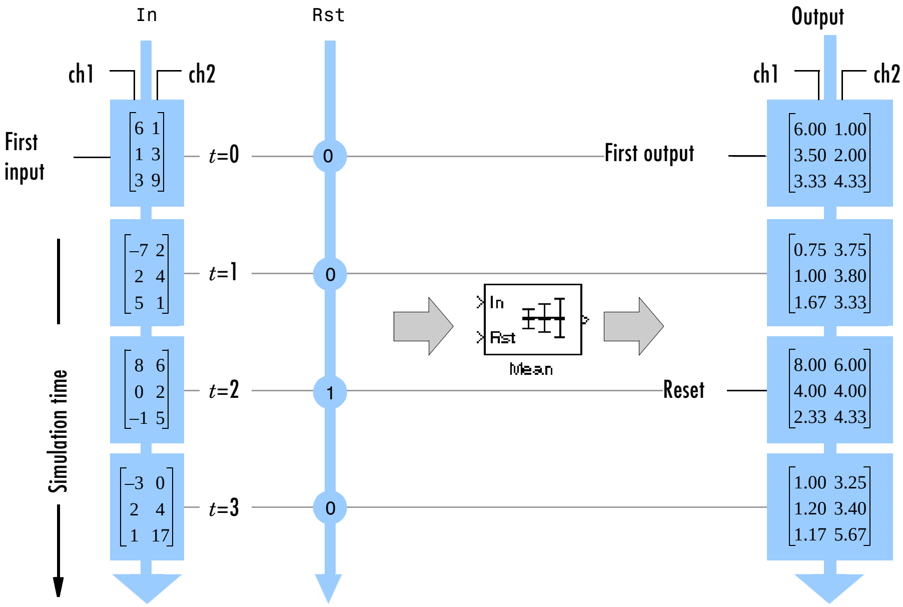
where

```
u = [6 1 3 -7 2 5 8 0 -1 -3 2 1;1 3 9 2 4 1 6 2 5 0 4 17]'
```

The Discrete Impulse block has the following settings:

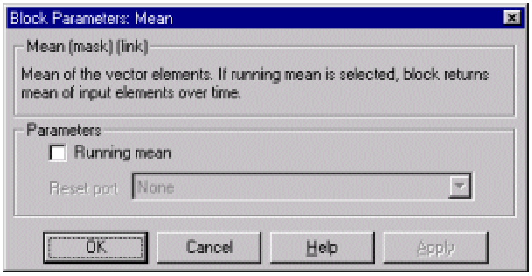
- **Delay (samples)** = `2`
- **Sample time** = `1`
- **Samples per frame** = `1`

The block's operation is shown in the figure below.



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

Dialog Box


Block Parameters: Mean

Mean (mask) (link)

Mean of the vector elements. If running mean is selected, block returns mean of input elements over time.

Parameters

□ Running mean

Reset port: None

Running mean

Enables running operation when selected.

Reset port

Determines the reset event that causes the block to reset the running mean. The rate of the reset signal must be a positive integer multiple of the rate of the data signal input. This parameter is enabled only when you set the **Running mean** parameter. For more information, see “Resetting the Running Histogram” on page 7-381.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Boolean — The block accepts Boolean inputs to the `Rst` port.

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

See Also

Maximum	DSP Blockset
Median	DSP Blockset
Minimum	DSP Blockset
Standard Deviation	DSP Blockset
`mean`	MATLAB

Median

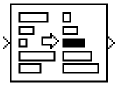
Purpose

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.

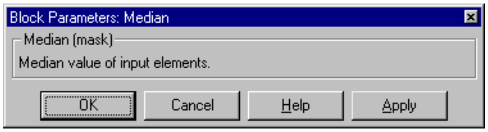
```
s = sort(u);
y = s((M+1)/2,:)
```

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



Block Parameters: Median

Median (mask)

Median value of input elements.

Supported Data Types

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

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

See Also

Maximum	DSP Blockset
Mean	DSP Blockset
Minimum	DSP Blockset
Sort	DSP Blockset
Standard Deviation	DSP Blockset
Variance	DSP Blockset
`median`	MATLAB

Minimum

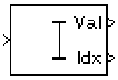
Purpose

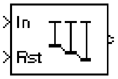
Find the minimum values in an input or sequence of inputs

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`.

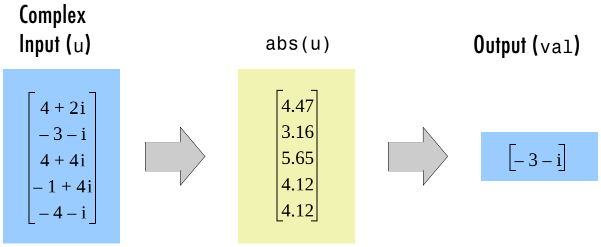
Value Mode

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.

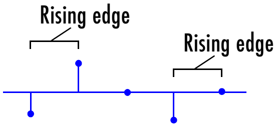
As in the other modes, length-M 1-D vector inputs and *sample-based* length-M row vector inputs are both treated as M-by-1 column vectors.

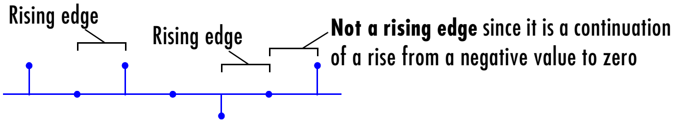
Resetting the Running Minimum. The block resets the running minimum whenever a reset event is detected at the optional `Rst` port. The rate of the reset signal must be a positive integer multiple of the rate of the data signal input.

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.

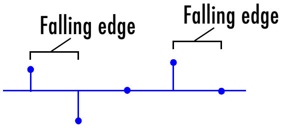
The reset event is specified by the **Reset port** parameter, and can be one of the following:

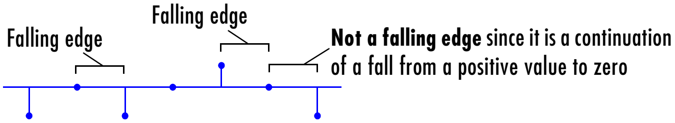
- `None` disables the `Rst` port.
- `Rising edge` — Triggers a reset operation when the `Rst` input does one of the following:
  - Rises from a negative value to a positive value or zero
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)




Rising edge
Rising edge
Not a rising edge since it is a continuation
of a rise from a negative value to zero

- `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)



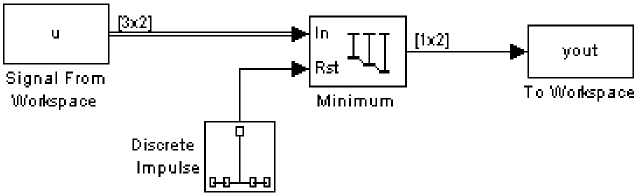

Falling edge
Falling edge
Not a falling edge since it is a continuation
of a fall from a positive value to zero

- `Either edge` — Triggers a reset operation when the `Rst` input is a `Rising edge` or `Falling edge` (as described above).
- `Non-zero sample` — Triggers a reset operation at each sample time that the `Rst` input is not zero.

Note When running simulations in the Simulink `MultiTasking` mode, sample-based reset signals have a one-sample latency, and frame-based reset signals have one frame of latency. Thus, there is a one-sample or one-frame delay between the time the block detects a reset event, and when it applies the reset. For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" on page 2-98 and the topic on The Simulation Parameters Dialog Box in the Simulink documentation.

Examples

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

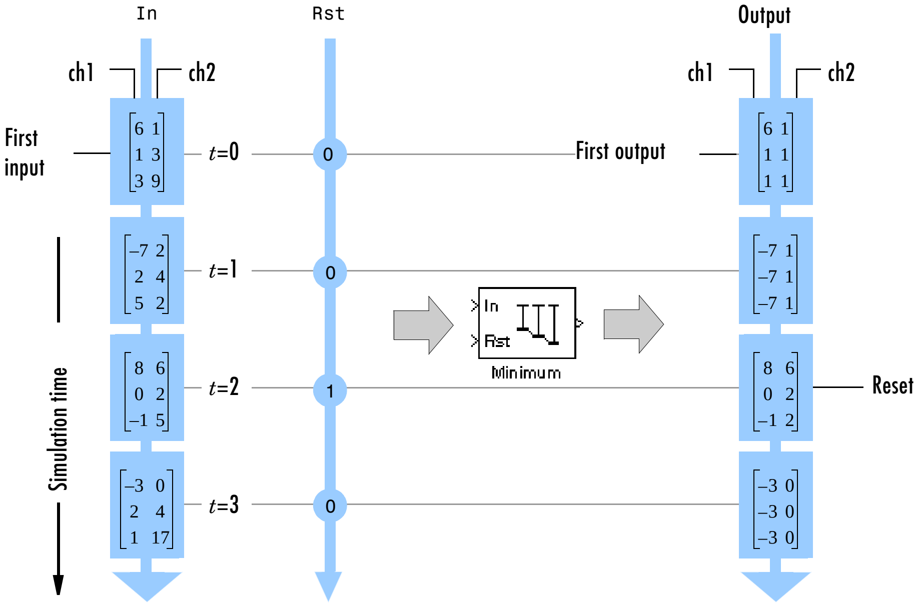
```
u = [6 1 3 -7 2 5 8 0 -1 -3 2 1;1 3 9 2 4 2 6 2 5 0 4 17]'
```

The Discrete Impulse block has the following settings:

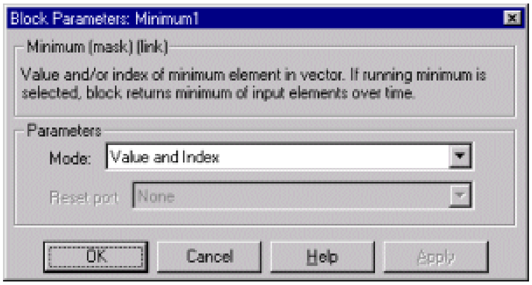
- **Delay (samples)** = `2`
- **Sample time** = `1`

- **Samples per frame** = 1

The block's operation is shown in the figure below.



Dialog Box


Block Parameters: Minimum1

Minimum (mask) (link)

Value and/or index of minimum element in vector. If running minimum is selected, block returns minimum of input elements over time.

Parameters

Mode: Value and Index

Reset port: None

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 7-539.

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

See Also

Maximum	DSP Blockset
Mean	DSP Blockset
MinMax	Simulink
Histogram	DSP Blockset
`min`	MATLAB

Modified Covariance AR Estimator

Purpose

Compute an estimate of AR model parameters using the modified covariance method

Library

Estimation / Parametric Estimation

Description

MCov AR
Estimator
A
G

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.

$$H(z) = \frac{G}{A(z)} = \frac{G}{1 + a(2)z^{-1} + \ldots + a(p+1)z^{-p}}$$

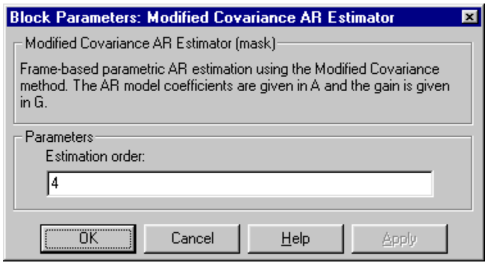
The order, p, of the all-pole model is specified by the **Order** parameter.

The output port labeled A outputs the normalized estimate of the AR model coefficients in descending powers of z.

```
[1 a(2) ... a(p+1)]
```

The scalar gain, G, is output from the output port labeled G.

Dialog Box


Block Parameters: Modified Covariance AR Estimator

Modified Covariance AR Estimator (mask)

Frame-based parametric AR estimation using the Modified Covariance method. The AR model coefficients are given in A and the gain is given in G.

Parameters

Estimation order: 4

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 your data types to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-2.

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

Modified Covariance Method

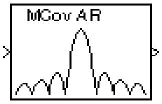
Purpose

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.

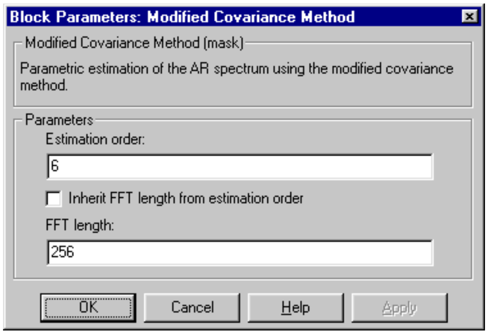
When **Inherit FFT length from input dimensions** is selected, N_{fft} is specified by the frame size of the input, which must be a power of 2. When **Inherit FFT length from input dimensions** is *not* selected, N_{fft} is specified as a power of 2 by the **FFT length** parameter, and the block zero 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.

Dialog Box



Block Parameters: Modified Covariance Method

Modified Covariance Method (mask)

Parametric estimation of the AR spectrum using the modified covariance method.

Parameters

Estimation order: 6

□ Inherit FFT length from estimation order

FFT length: 256

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

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 5-5 for related information.

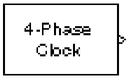
Purpose

Generate multiple binary clock signals

Library

- DSP Sources
- Signal Management / Switches and Counters

Description



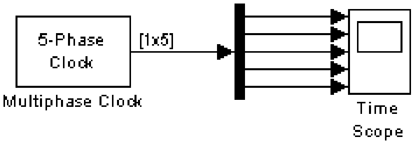
The Multiphase Clock block generates a sample-based 1-by-N vector of clock signals, where the integer N is specified by the **Number of phases** parameter. Each of the N phases has the same frequency, *f*, specified in hertz by the **Clock frequency** parameter.

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.

Examples

Configure the Multiphase Clock block in the model below to generate a 100 Hz five-phase output in which the third signal is first to become active. Use a *high* active level with a duration of one interval.

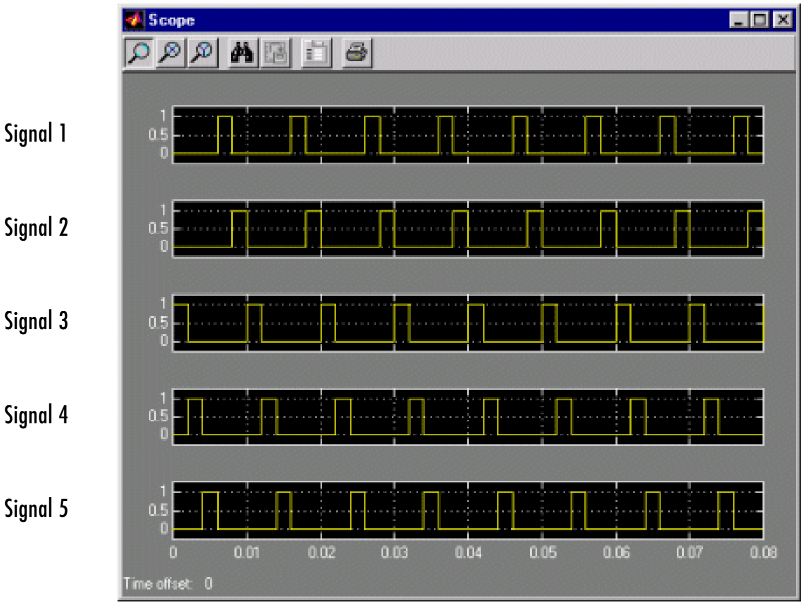


The corresponding settings are as follows:

- **Clock frequency** = 100
- **Number of phases** = 5
- **Starting phase** = 3
- **Number of phase intervals over which the clock is active** = 1

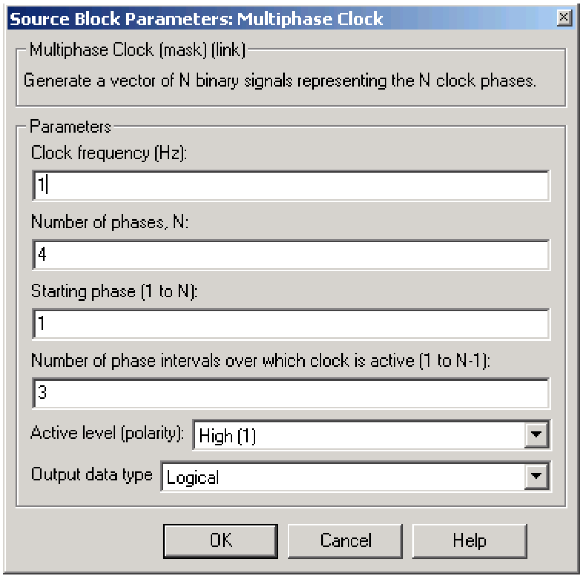
- **Active level (polarity)** = `High (1)`

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



Source Block Parameters: Multiphase Clock

Multiphase Clock (mask) (link)

Generate a vector of N binary signals representing the N clock phases.

Parameters

Clock frequency (Hz): 1

Number of phases, N: 4

Starting phase (1 to N): 1

Number of phase intervals over which clock is active (1 to N-1): 3

Active level (polarity): High (1)

Output data type Logical

Opening this dialog box causes a running simulation to pause. See “Changing Source Block Parameters” in the online Simulink documentation for details.

Clock frequency

The frequency of all output clock signals.

Number of phases

The number of different phases, N, in the output vector.

Starting phase

The vector index of the output signal to first become active. Tunable.

Number of phase intervals over which clock is active

The duration of the active level for every output signal. Tunable in simulation, but not in Real-Time Workshop external mode.

Active level

The active level, `High (1)` or `Low (0)`. Tunable.

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-7.

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-7. To learn how to disable Boolean output support, see “Steps to Disabling Boolean Support” on page A-8.

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

See Also

Clock	Simulink
Counter	DSP Blockset
Pulse Generator	Simulink
Event-Count Comparator	DSP Blockset

Also see “Creating Signals Using Signal Generator Blocks” on page 2-35 for how to use this and other blocks to generate signals.

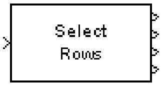
Purpose

Distribute arbitrary subsets of input rows or columns to multiple output ports

Library

Signal Management / Indexing

Description



The Multiport Selector block extracts multiple subsets of rows or columns from M-by-N input matrix *u*, and propagates each new submatrix to a distinct output port. A length-M 1-D vector input is treated as an M-by-1 matrix.

The **Indices to output** parameter is a cell array whose *k*th cell contains a one-dimensional indexing expression specifying the subset of input rows or columns to be propagated to the *k*th output port. The total number of cells in the array determines the number of output ports on the block.

When the **Select** parameter is set to `Rows`, the specified one-dimensional indices are used to select matrix rows, and all elements on the chosen rows are included. When the **Select** parameter is set to `Columns`, the specified one-dimensional indices are used to select matrix columns, and all elements on the chosen columns are included. A given input row or column can appear any number of times in any of the outputs, or not at all.

When an index references a nonexistent row or column of the input, the block reacts with the behavior specified by the **Invalid index** parameter. The following options are available:

- `Clip index` — Clip the index to the nearest valid value, and *do not* issue an alert.

  Example: For a 64-by-4 input with **Select** = `Rows`, an index of 72 is clipped to 64; with **Select** = `Columns`, an index of 72 is clipped to 4. In both cases, an index of -2 is clipped to 1.
- `Clip and warn` — Display a warning message in the MATLAB Command Window, and clip as above.
- `Generate error` — Display an error dialog box and terminate the simulation.

Examples

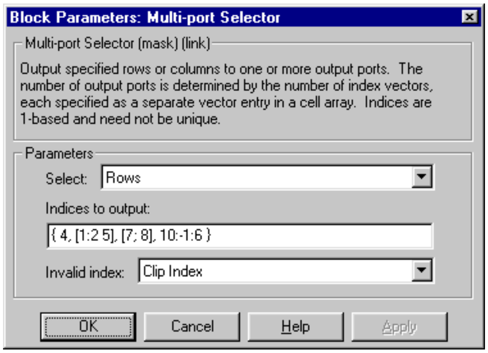
Consider the following **Indices to output** cell array:

```
{4,[1:2 5],[7;8],10:-1:6}
```

This is a four-cell array, which requires the block to generate four independent outputs (each at a distinct port). The table below shows the dimensions of these outputs when **Select** = `Rows` and the input dimension is M-by-N.

Cell	Expression	Description	Output Size
1	`4`	Row 4 of input	1-by-N
2	`[1:2 5]`	Rows 1, 2, and 5 of input	3-by-N
3	`[7;8]`	Rows 7 and 8 of input	2-by-N
4	`10:-1:6`	Rows 10, 9, 8, 7, and 6 of input	5-by-N

Dialog Box



Block Parameters: Multi-port Selector

Multi-port Selector (mask) (link)

Output specified rows or columns to one or more output ports. The number of output ports is determined by the number of index vectors, each specified as a separate vector entry in a cell array. Indices are 1-based and need not be unique.

Parameters

Select: Rows

Indices to output:

{ 4, [1:2 5], [7; 8], 10:-1:6 }

Invalid index: Clip Index

Select

The dimension of the input to select, `Rows` or `Columns`.

Indices to output

A cell array specifying the row- or column-subsets to propagate to each of the output ports. The number of cells in the array determines the number of output ports on the block.

Invalid index

Response to an invalid index value. 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 your data types to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-2.

See Also

Permute Matrix	DSP Blockset
Selector	Simulink
Submatrix	DSP Blockset
Variable Selector	DSP Blockset

N-Sample Enable

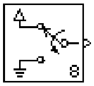
Purpose

Output ones or zeros for a specified number of sample times

Library

- DSP Sources
- Signal Management / Switches and Counters

Description



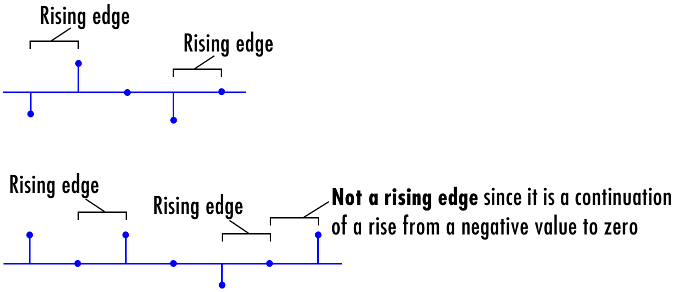
The N-Sample Enable block outputs the *inactive* value (`0` or `1`, whichever is *not* selected in the **Active level** parameter) during the first N sample times, where N is the **Trigger count** value. Beginning with output sample N+1, the block outputs the *active* value (`1` or `0`, whichever is 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. This block supports triggered subsystems when the **Reset input** check box is selected.

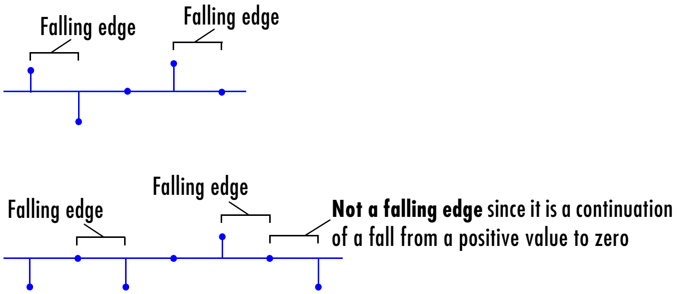
The triggering event is specified by the **Trigger type** pop-up menu, and can be one of the following:

- `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)


Rising edge
Rising edge
Rising edge
Rising edge
Not a rising edge since it is a continuation of a rise from a negative value to zero

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

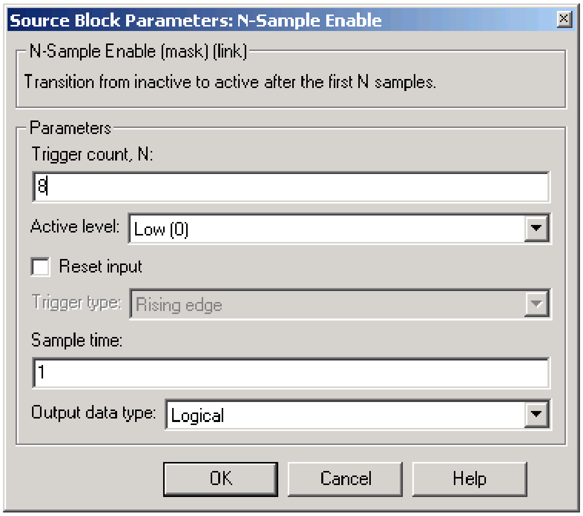
  - Falls from a positive value to a negative value or zero
  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero (see the following figure)


Falling edge
Falling edge
Falling edge
Falling edge
Not a falling edge since it is a continuation of a fall from a positive value to zero

- `Either edge` — Triggers a reset operation when the `Rst` input is a `Rising edge` or `Falling edge` (as described above).
- `Non-zero sample` — Triggers a reset operation at each sample time that the `Rst` input is not zero.

Note When running simulations in the Simulink `MultiTasking` mode, sample-based reset signals have a one-sample latency, and frame-based reset signals have one frame of latency. Thus, there is a one-sample or one-frame delay between the time the block detects a reset event, and when it applies the reset. For more information on latency and the Simulink tasking modes, see “Excess Algorithmic Delay (Tasking Latency)” on page 2-98 and the topic on The Simulation Parameters Dialog Box in the Simulink documentation.

Dialog Box



Source Block Parameters: N-Sample Enable

N-Sample Enable (mask) (link)

Transition from inactive to active after the first N samples.

Parameters

Trigger count, N: 8

Active level: Low (0)

□ Reset input

Trigger type: Rising edge

Sample time: 1

Output data type: Logical

Opening this dialog box causes a running simulation to pause. See “Changing Source Block Parameters” in the online Simulink documentation for details.

Trigger count

The number of samples for which the block outputs the active value. Tunable.

Active level

The value to output after the first N sample times, `0` or `1`. Tunable.

Reset input

Enables the `Rst` input port. The rate of the reset signal must be a positive integer multiple of the rate of the data signal input.

Trigger type

The type of event that triggers a reset when the `Rst` port is enabled. Nontunable.

Sample time

The sample period, T_s, for the block’s counter. The block switches from the active value to the inactive value at t=T_s*(N+1).

Output data type

The output data type. Nontunable. For information on the `Logical` and `Boolean` options of this parameter, see “Effects of Enabling and Disabling Boolean Support” on page A-7.

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-7. To learn how to disable Boolean output support, see “Steps to Disabling Boolean Support” on page A-8.

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

See Also

Counter	DSP Blockset
N-Sample Switch	DSP Blockset

Also see “Creating Signals Using Signal Generator Blocks” on page 2-35 for how to use this and other blocks to generate signals.

N-Sample Switch

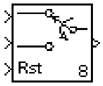
Purpose

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 (that is, the output sample period is not inherited from the sample period of either input). The block applies a zero-order hold at the input ports, so the value the block reads from a given port between input sample times is the value of the most recent input to that port.

Both inputs must have the same dimension, except in the following two cases:

- When one input is a scalar, the block expands the scalar input to match the size of the other input.
- When one input is a 1-D vector and the other input is a row or column vector with the same number of elements, the block reshapes the 1-D vector to match the dimension of the other input.

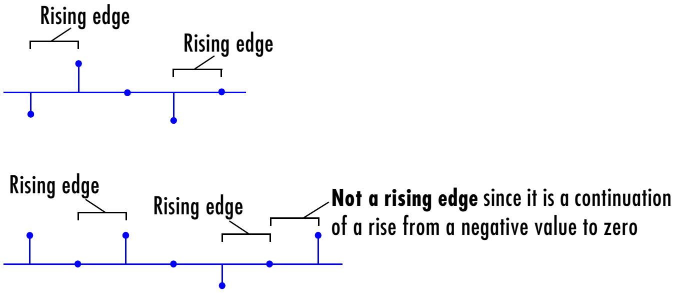
The inputs must either both be frame based or both be sample based.

The **Reset input** check box enables the `Rst` input port. At any time during the count, a trigger event at the `Rst` port resets the counter to zero. The rate of the reset signal must be a positive integer multiple of the rate of the data signal input. This block supports triggered subsystems when the **Reset input** check box is selected.

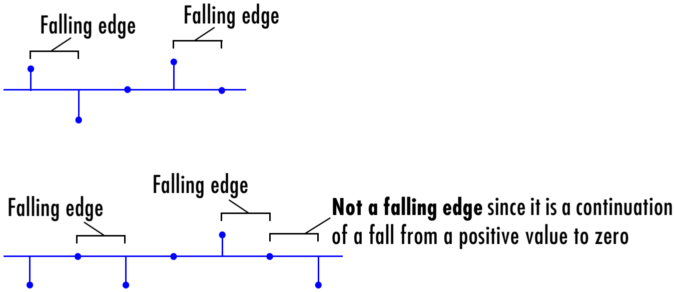
The triggering event is specified by the **Trigger type** pop-up menu, and can be one of the following:

- `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)


Rising edge
Rising edge
Rising edge
Rising edge
Not a rising edge since it is a continuation of a rise from a negative value to zero

- `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)

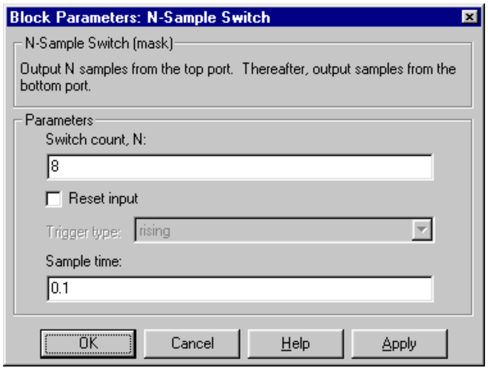

Falling edge
Falling edge
Falling edge
Falling edge
Not a falling edge since it is a continuation of a fall from a positive value to zero

- `Either edge` — Triggers a reset operation when the `Rst` input is a `Rising edge` or `Falling edge` (as described above).
- `Non-zero sample` — Triggers a reset operation at each sample time that the `Rst` input is not zero.

Note When running simulations in the Simulink `MultiTasking` mode, sample-based reset signals have a one-sample latency, and frame-based reset signals have one frame of latency. Thus, there is a one-sample or one-frame delay between the time the block detects a reset event, and when it applies the reset. For more information on latency and the Simulink tasking modes, see

“Excess Algorithmic Delay (Tasking Latency)” on page 2-98 and the topic on The Simulation Parameters Dialog Box in the Simulink documentation.

Dialog Box



Block Parameters: N-Sample Switch

N-Sample Switch (mask)

Output N samples from the top port. Thereafter, output samples from the bottom port.

Parameters

Switch count, N: 8

□ Reset input

Trigger type: rising

Sample time: 0.1

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

See Also

Counter	DSP Blockset
N-Sample Enable	DSP Blockset

Normalization

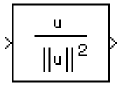
Purpose

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 (for example, 1e-10) that prevents potential division by zero.

Squared 2-Norm

When the **Norm** parameter specifies **Squared 2-norm**, the block normalizes the *j*th input column as follows

$$y_{ij} = \frac{u_{ij}}{\|u\|_j^2 + b}$$

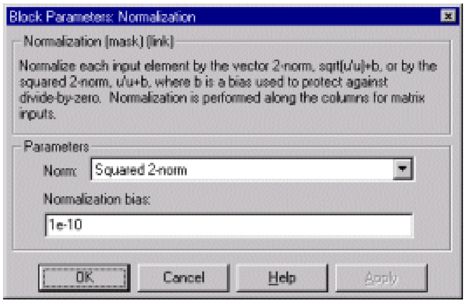
where

$$\|u\|_j^2 = |u_1|^2 + |u_2|^2 + \cdots + |u_M|^2$$

Equivalently,

```
y = u ./ (norm(u).^2 + b)   % Equivalent MATLAB code
```

Dialog Box



Normalize each input element by the vector 2-norm, sqrt(u'u)+b, or by the squared 2-norm, u'u+b, where b is a bias used to protect against divide-by-zero. Normalization is performed along the columns for matrix inputs.

Norm: Squared 2-norm

Normalization bias: 1e-10

Norm

The type of normalization to apply, `2-norm` or `Squared 2-norm`. Tunable.

Normalization bias

The real value *b* to be added in the denominator to avoid division by zero. Tunable.

Supported Data Types

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

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

See Also

Matrix Scaling	DSP Blockset
Reciprocal Condition	DSP Blockset
`norm`	MATLAB

Purpose

Implement the overlap-add method of frequency-domain filtering

Library

Filtering / Filter Designs

Description

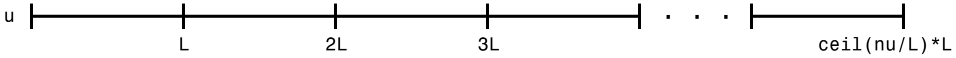
The Overlap-Add FFT Filter block uses an FFT to implement the *overlap-add method*, a technique that combines successive frequency-domain filtered sections of an input sequence.

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.

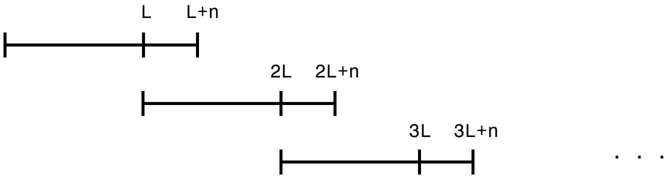
If either the filter coefficients or the inputs to the block are complex, the **Output** parameter should be set to `Complex`. Otherwise, the default **Output** setting, `Real`, instructs the block to take only the real part of the solution.

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
```

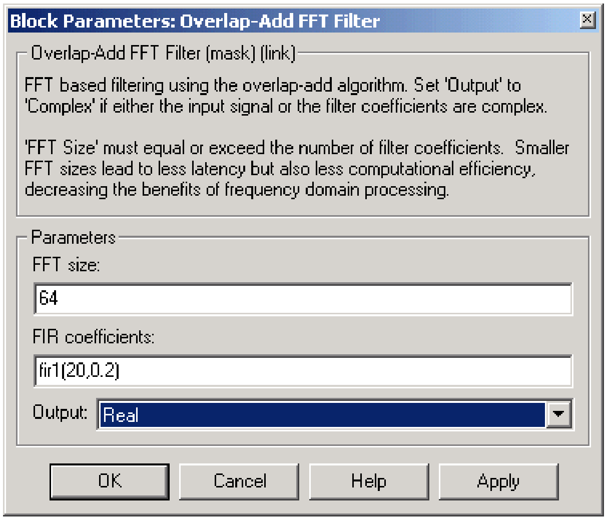
Latency

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 2-98 and the topic called The Simulation Parameters Dialog Box in the Simulink documentation for more information about block rates and the Simulink tasking modes.

Dialog Box



Block Parameters: Overlap-Add FFT Filter

Overlap-Add FFT Filter (mask) (link)

FFT based filtering using the overlap-add algorithm. Set 'Output' to 'Complex' if either the input signal or the filter coefficients are complex.

'FFT Size' must equal or exceed the number of filter coefficients. Smaller FFT sizes lead to less latency but also less computational efficiency, decreasing the benefits of frequency domain processing.

Parameters

FFT size: 64

FIR coefficients: fir1(20,0.2)

Output: Real

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

See Also

Overlap-Save FFT Filter	DSP Blockset

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.

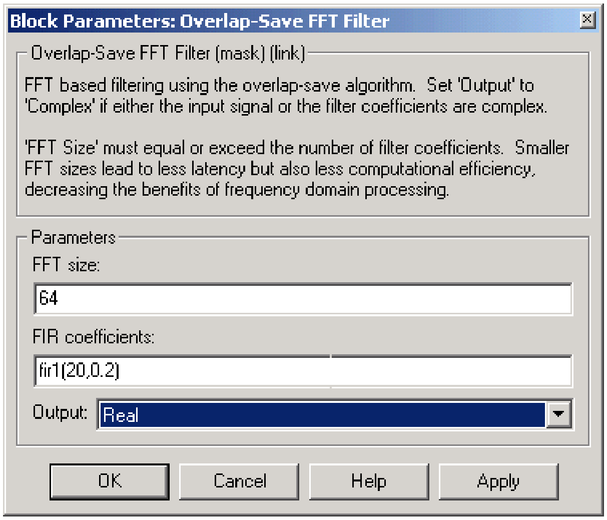
Latency

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 2-98 and the topic called The Simulation Parameters Dialog Box in the Simulink documentation for more information about block rates and the Simulink tasking modes.

Dialog Box



Block Parameters: Overlap-Save FFT Filter

Overlap-Save FFT Filter (mask) (link)

FFT based filtering using the overlap-save algorithm. Set 'Output' to 'Complex' if either the input signal or the filter coefficients are complex.

'FFT Size' must equal or exceed the number of filter coefficients. Smaller FFT sizes lead to less latency but also less computational efficiency, decreasing the benefits of frequency domain processing.

Parameters

FFT size: 64

FIR coefficients: fir1(20,0.2)

Output: Real

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 your data types to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-2.

See Also

Overlap-Add FFT Filter	DSP Blockset

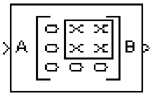
Purpose

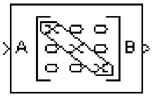
Overwrite a submatrix or subdiagonal of the input.

Library

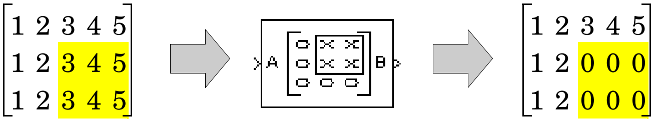
- Math Functions / Matrices and Linear Algebra / Matrix Operations
- Signal Management / Indexing

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

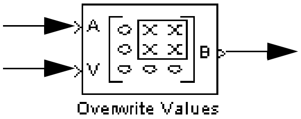
Specifying the Overwriting Values

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 7-576.
- `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.)

Input signal

Signal of overwriting values



The output is the result of overwriting the **Input signal**, and has the same size, rate, and frame status as the **Input signal**.

The rate at which you provide the overwriting values through input port `V` must match the rate at which the block receives each input matrix at input port `A`. The rate requirements depend on whether the input signal and overwriting values signal have the same frame status:

- 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.

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	$k = 3 \qquad v = \begin{bmatrix} 2 & 4 & 6 \end{bmatrix}$ 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	$k = 3 \qquad v = \begin{bmatrix} 2 & 4 & 6 \end{bmatrix}$ $\begin{bmatrix} x & x & x & x & x \\ x & 2 & 4 & 6 & x \\ x & x & x & x & x \\ x & x & x & x & x \end{bmatrix}$

Valid Overwriting Values (Continued)

Portion of Input to Overwrite	Valid Overwriting Values	Example
A length-k portion of a column $\begin{bmatrix} x & x & x & x & x \\ x & x & x & \colorbox{yellow}{x} & x \\ x & x & x & \colorbox{yellow}{x} & x \\ x & x & x & x & x \end{bmatrix}$	Any length-k column vector, v	$k = 2 \qquad v = \begin{bmatrix} 4 \\ 6 \end{bmatrix}$ $\begin{bmatrix} x & x & x & x & x \\ x & x & x & \colorbox{yellow}{4} & x \\ x & x & x & \colorbox{yellow}{6} & x \\ x & x & x & x & x \end{bmatrix}$
An m-by-n submatrix $\begin{bmatrix} x & x & x & x & x \\ x & x & \colorbox{yellow}{x} & \colorbox{yellow}{x} & \colorbox{yellow}{x} \\ x & x & \colorbox{yellow}{x} & \colorbox{yellow}{x} & \colorbox{yellow}{x} \\ x & x & x & x & x \end{bmatrix}$	Any m-by-n matrix, v	$m = 2$ $n = 3 \qquad v = \begin{bmatrix} 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$ $\begin{bmatrix} x & x & x & x & x \\ x & x & \colorbox{yellow}{4} & \colorbox{yellow}{5} & \colorbox{yellow}{6} \\ x & x & \colorbox{yellow}{7} & \colorbox{yellow}{8} & \colorbox{yellow}{9} \\ x & x & x & x & x \end{bmatrix}$

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 7-576.
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 the following tables.

Settings for Row, Column, Starting Row, and Starting Column Parameters

Settings for Specifying the Submatrix's First Row or Column	**First Row of Submatrix** (Only row for **Row span** = **One row**)	**First Column of Submatrix** (Only row for **Row span** = **One row**)
`First`	First row of the input	First column of the input
`Index`	Input row specified in the **Row index** parameter	Input column specified in the **Column index** parameter

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

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 7-576.

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 the following tables.

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

Ending Element Parameters

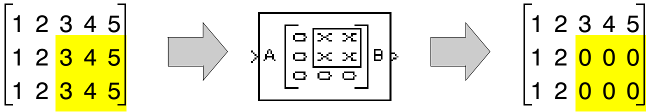
Settings for Ending Element Parameter	Last Element in Subdiagonal
`Index`	*k*th diagonal element, where *k* is the value of the **Ending element index** parameter
`Offset from last`	Diagonal element in the row with the index `M – offset` where `M` is the number of input rows, and `offset` is the value 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`

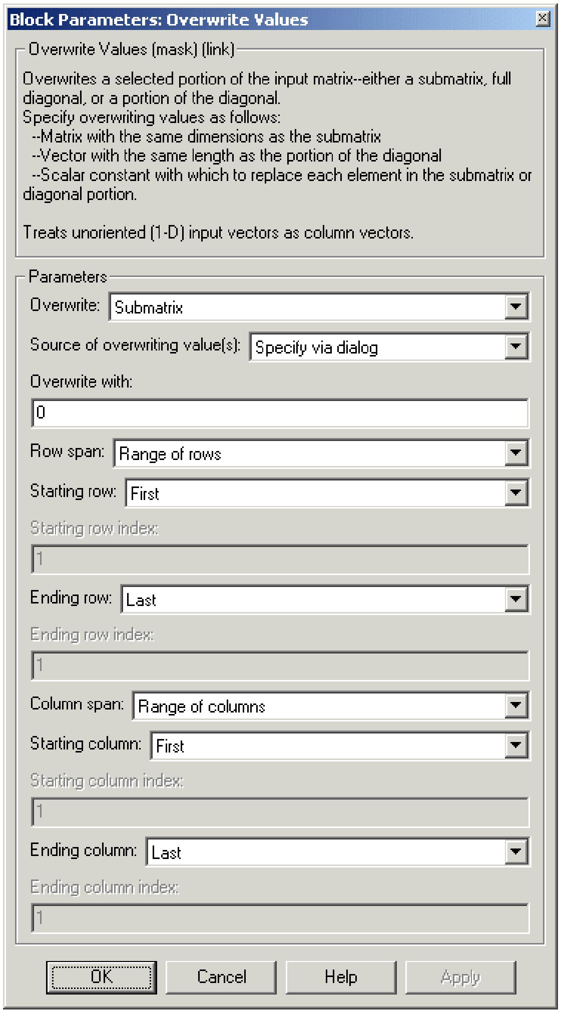
The figure below shows the block with the above settings overwriting a portion of a 3-by-5 input matrix.



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



Block Parameters: Overwrite Values

Overwrite Values (mask) (link)

Overwrites a selected portion of the input matrix--either a submatrix, full diagonal, or a portion of the diagonal.
Specify overwriting values as follows:
 --Matrix with the same dimensions as the submatrix
 --Vector with the same length as the portion of the diagonal
 --Scalar constant with which to replace each element in the submatrix or diagonal portion.

Treats unoriented (1-D) input vectors as column vectors.

Parameters

Overwrite: Submatrix

Source of overwriting value(s): Specify via dialog

Overwrite with: 0

Row span: Range of rows

Starting row: First

Starting row index: 1

Ending row: Last

Ending row index: 1

Column span: Range of columns

Starting column: First

Starting column index: 1

Ending column: Last

Ending column index: 1

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 7-576.

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 7-576.

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 7-579.

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 the table called "Settings for Row, Column, Starting Row, and Starting Column Parameters" on page 7-579. (**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 the table called "Settings for Row, Column, Starting Row, and Starting Column Parameters" on page 7-579. (**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 the table called "Settings for Row, Column, Starting Row, and Starting Column Parameters" on page 7-579. (**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 the table called "Settings for Ending Row and Ending Column Parameters" on page 7-581. (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 the table called "Settings for Ending Row and Ending Column Parameters" on page 7-581. (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 the table called "Settings for Ending Row and Ending Column Parameters" on page 7-581. (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 7-579.

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 the table called "Settings for Row, Column, Starting Row, and Starting Column Parameters" on page 7-579. (**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 the table called

"Settings for Row, Column, Starting Row, and Starting Column Parameters" on page 7-579. (**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 the table called "Settings for Row, Column, Starting Row, and Starting Column Parameters" on page 7-579. (**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 in the table called "Settings for Ending Row and Ending Column Parameters" on page 7-581. (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 the table called "Settings for Ending Row and Ending Column Parameters" on page 7-581. (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 the table called "Settings for Ending Row and Ending Column Parameters" on page 7-581. (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 7-581.

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 the table called “Element and Starting Element Parameters” on page 7-582. (**Element** is enabled when **Element span** is set to `One element`, and **Starting element** when **Element span** is set to `Range of elements`.)

Element index/Starting element index

Index of the input diagonal element that is the first element of the subdiagonal that the block overwrites. See how to use these parameters in the table called “Element and Starting Element Parameters” on page 7-582. (**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 the table called “Element and Starting Element Parameters” on page 7-582. (**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 the table called “Ending Element Parameters” on page 7-583. (Enabled when **Element span** is set to `Range of elements`, and **Starting element** is set to any option but `Last`.)

Ending element index

Index of the input diagonal element that is the last element of the subdiagonal that the block overwrites. See how to use this parameter in the table called “Ending Element Parameters” on page 7-583. (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 the table called “Ending Element Parameters” on page 7-583. (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 your data types to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-2.

See Also

Reshape	Simulink
Selector	Simulink
Submatrix	DSP Blockset
Variable Selector	DSP Blockset
`reshape`	MATLAB

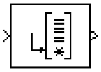
Purpose

Alter the input dimensions by padding (or truncating) rows and/or columns

Library

Signal Operations

Description



Use the **Value** parameter to specify the value with which to pad your input matrix.

Using the **Pad signal at** parameter, you can choose to pad your input matrix at the end or the beginning of a row and/or column.

The Pad block changes the dimensions of the input matrix from M_i-by-N_i to M_o-by-N_o by padding or truncating along the columns, rows, or columns and rows. Use the **Pad along** parameter to specify the dimensions to change.

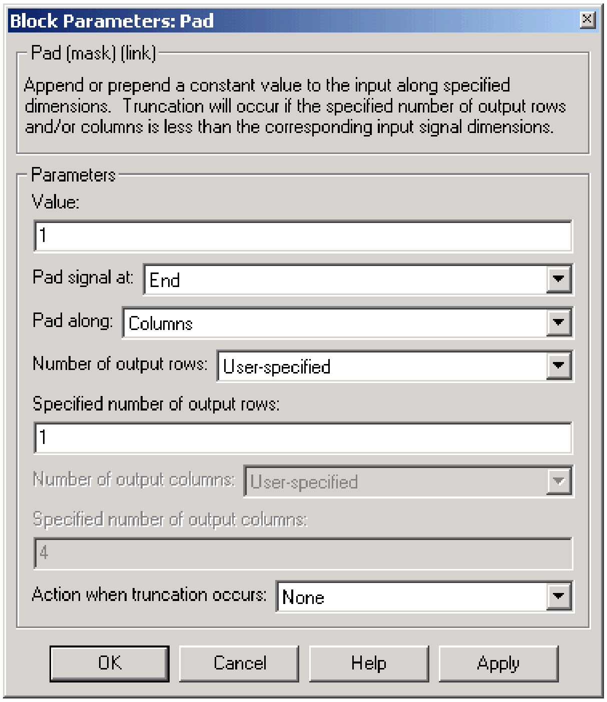
The **Number of output rows** and/or **Number of output columns** parameters refer to the dimensions of the output, M_o and N_o. You can set these parameters to `User-specified` or `Next power of two`. If you choose `User-specified`, enter a scalar value in the **Specified number of output rows** and/or **Specified number of output columns** parameters. If you choose `Next power of two`, the block pads the input matrix along the columns and/or rows until the length of the columns and/or rows is equal to a power of two. If the length of the input matrix's columns and/or rows is already equal to a power of two, the block does not pad the input matrix.

If you choose `User-specified` for the **Number of output rows** and/or **Number of output columns** parameters, you can specify a scalar value in the **Specified number of output rows** and/or **Specified number of output columns** parameters that truncates the size of your input matrix. The following options are available for the **Action when truncation occurs** parameter:

- `None` — Select this option if you do not want to be notified that the input matrix is truncated.
- `Warning` — Choose this option if you want a warning to be displayed in the MATLAB Command Window when the input matrix is truncated.
- `Error` — Click this option if you want an error dialog box to be displayed and the simulation terminated when the input matrix is truncated.

The behavior of the Pad block and Zero Pad block is identical, with the exception that the Pad block can pad the input matrix with values other than zero. See the Zero Pad block reference page for more information on the behavior of the Zero Pad block.

Dialog Box



Block Parameters: Pad

Pad (mask) (link)

Append or prepend a constant value to the input along specified dimensions. Truncation will occur if the specified number of output rows and/or columns is less than the corresponding input signal dimensions.

Parameters

Value: 1

Pad signal at: End

Pad along: Columns

Number of output rows: User-specified

Specified number of output rows: 1

Number of output columns: User-specified

Specified number of output columns: 4

Action when truncation occurs: None

Value

The scalar value with which to pad the input matrix. Tunable.

Pad signal at

The input matrix can be padded at the beginning of the rows and/or columns or at the end of the rows and/or columns.

Pad along

The direction along which to pad or truncate. `Columns` specifies that the *row* dimension should be changed to M_o. `Rows` specifies that the *column* dimension should be changed to N_o. `Columns and rows` specifies that both column and row dimensions should be changed. None disables padding and truncation and passes the input through to the output unchanged.

Number of output rows

The total number of output rows. If you select User-specified, type a scalar value in the **Specified Number of output rows** parameter. If you select Next power of two, the block pads the columns of the input matrix until the number of rows is equal to a power of two. If the number of rows is already equal to a power of two, the block does not pad the input matrix.

Specified number of output rows

The desired number of rows in the output, M_o. This parameter is enabled when Columns or Columns and rows is selected in the **Pad along** menu and User-specified is chosen in the **Number of output rows** parameter.

Number of output columns

The total number of output columns. If you select User-specified, type a scalar value in the **Specified Number of output columns** parameter. If you select Next power of two, the block pads the rows of the input matrix until the number of columns is equal to a power of two. If the number of columns is already equal to a power of two, the block does not pad the input matrix.

Specified number of output columns

The desired number of columns in the output, N_o. This parameter is enabled when Rows or Columns and rows is selected in the **Pad along** menu and User-specified is chosen in the **Number of output columns** parameter.

Action when truncation occurs

Choose None if you do not want to be notified that the input matrix is truncated. Select Warning to display a warning when the input matrix is truncated. Choose Error if you want an error dialog box to be displayed and the simulation terminated when the input matrix is truncated.

Supported Data Types

- Double-precision floating-point
- Single-precision floating-point
- Fixed point
- Boolean

- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers
- Custom data types

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

See Also

Matrix Concatenation	Simulink
Repeat	DSP Blockset
Submatrix	DSP Blockset
Upsample	DSP Blockset
Variable Selector	DSP Blockset
Zero Pad	DSP Blockset

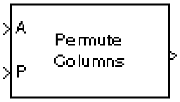
Purpose

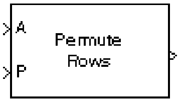
Reorder the rows or columns of a matrix

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:

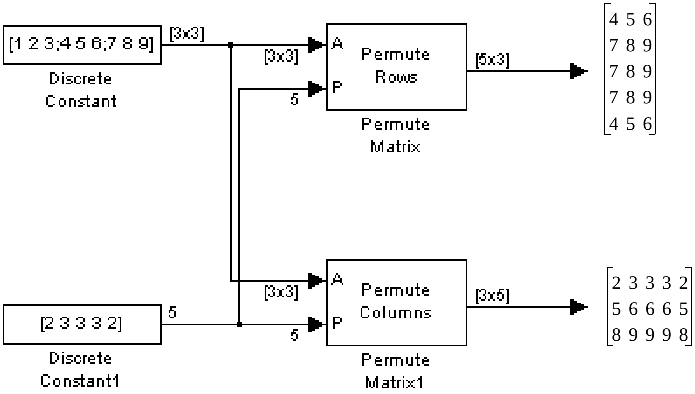
- Clip index — Clip the index to the nearest valid value (1 or M for row permutation, and 1 or N for column permutation), and *do not* issue an alert. Example: For a 3-by-7 input matrix, a column index of 9 is clipped to 7, and a row index of -2 is clipped to 1.
- Clip and warn — Display a warning message 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 selecting **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.

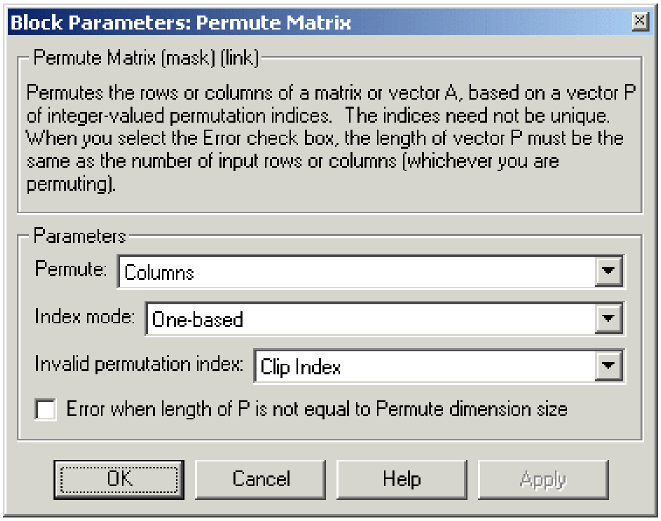
Examples

In the model below, the top Permute Matrix block places the second row of the input matrix in both the first and fifth rows of the output matrix, and places the third row of the input matrix in the three middle rows of the output matrix. The bottom Permute Matrix block places the second column of the input matrix in both the first and fifth columns of the output matrix, and places the third column of the input matrix in the three middle columns of the output matrix.



As shown in the example above, rows and columns of A can appear any number of times in the output, or not at all.

Dialog Box



Block Parameters: Permute Matrix

Permute Matrix (mask) (link)

Permutes the rows or columns of a matrix or vector A, based on a vector P of integer-valued permutation indices. The indices need not be unique. When you select the Error check box, the length of vector P must be the same as the number of input rows or columns (whichever you are permuting).

Parameters

Permute: Columns

Index mode: One-based

Invalid permutation index: Clip Index

□ Error when length of P is not equal to Permute dimension size

Permute

Method of constructing the output matrix; by permuting rows or columns of the input.

Index mode

When set to `One-based`, a value of `1` in the permutation vector `P` refers to the first row or column of the input matrix `A`. When set to `Zero-based`, a value of `0` in `P` refers to the first row or column of `A`.

Invalid permutation index

Response to an invalid index value. Tunable.

Error when length of P is not equal to Permute dimension size

Option to display an error dialog box and terminate the simulation 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 your data types to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-2.

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 2-68 for related information.

Purpose

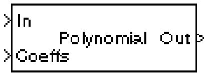
Evaluate a polynomial expression

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.

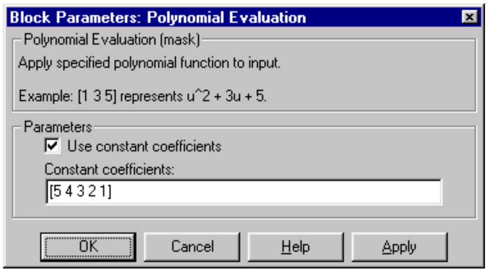
When the **Use constant coefficients** check box is selected, the polynomial expression is specified by the **Constant coefficients** parameter. When **Use constant coefficients** is not selected, a variable polynomial expression is specified by the input to the `Coeffs` port. In both cases, the polynomial 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



Block Parameters: Polynomial Evaluation

Polynomial Evaluation (mask)

Apply specified polynomial function to input.

Example: [1 3 5] represents u^2 + 3u + 5.

Parameters

☑ Use constant coefficients

Constant coefficients:

[5 4 3 2 1]

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 your data types to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-2.

See Also

Least Squares Polynomial Fit	DSP Blockset
Math Function	Simulink
Sum	Simulink
`polyval`	MATLAB

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,

$$f(x) = u_1x^{M-1} + u_2x^{M-2} + \cdots + u_M$$

arranged in order of descending exponents, u_1, u_2, ..., u_M. The polynomial has order M-1 and positive integer exponents.

Inputs can be frame based or sample based, and both represent the polynomial coefficients as shown above. For convenience, a length-M 1-D vector input is treated as an M-by-1 matrix.

The output is a 1-by-N matrix with each column containing the value 1 or 0. The value 1 indicates that the polynomial in the corresponding column of the input is stable; that is, the magnitudes of all solutions to $f(x) = 0$ are less than 1. The value 0 indicates that the polynomial in the corresponding column of the input may be unstable; that is, the magnitude of at least one solution to $f(x) = 0$ is greater than or equal to 1.

The output is always sample based.

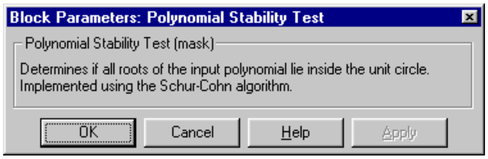
Applications

This block is most commonly used to check the pole locations of the denominator polynomial, $A(z)$, of a transfer function, $H(z)$.

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_1 + b_2z^{-1} + \ldots + b_mz^{-(m-1)}}{a_1 + a_2z^{-1} + \ldots + a_nz^{-(n-1)}}$$

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



Block Parameters: Polynomial Stability Test

Polynomial Stability Test (mask)

Determines if all roots of the input polynomial lie inside the unit circle. Implemented using the Schur-Cohn algorithm.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Boolean — Block outputs are always Boolean.

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

See Also

Least Squares Polynomial Fit	DSP Blockset
Polynomial Evaluation	DSP Blockset
`polyfit`	MATLAB

Purpose

Compute the Moore-Penrose pseudoinverse of a matrix

Library

Math Functions / Matrices and Linear Algebra / Matrix Inverses

Description

Pseudoinverse
(SVD)

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

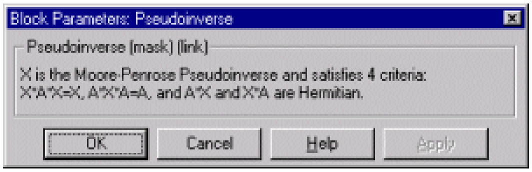
$$A^+ = VS^+U^*$$

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



Pseudoinverse (mask) (link)

X is the Moore-Penrose Pseudoinverse and satisfies 4 criteria:
X*A*X=X, A*X*A=A, and A*X and X*A are Hermitian.

References

Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

Supported Data Types

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

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

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 5-9 for related information.

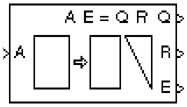
Purpose

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 a modified Gram-Schmidt iteration to factor a column permutation of the M-by-N input matrix A as

$$A_e = QR$$

where Q is an M-by-min(M,N) unitary matrix, and R is a min(M,N)-by-N upper-triangular matrix. A length-M vector input is treated as an M-by-1 matrix, and is always sample based.

The column-pivoted matrix A_e contains the columns of A permuted as indicated by the contents of length-N permutation vector E.

```
Ae = A(:,E)        % Equivalent MATLAB code
```

The block selects a column permutation vector E, which ensures that the diagonal elements of matrix R are arranged in order of decreasing magnitude.

$$|r_{i+1,j+1}| > |r_{i,j}| \qquad i = j$$

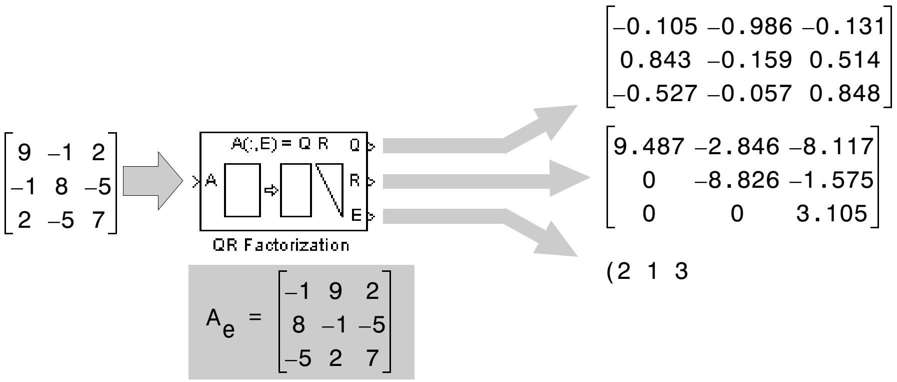
QR factorization is an important tool for solving linear systems of equations because of good error propagation properties and the invertability of unitary matrices.

$$Q^{-1} = Q^{*}$$

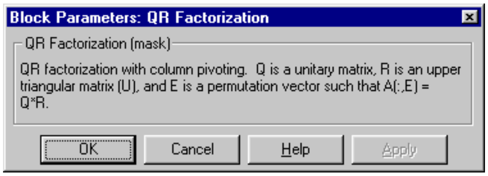
Unlike LU and Cholesky factorizations, the matrix A does not need to be square for QR factorization. Note, however, that QR factorization requires twice as many operations as Gaussian elimination.

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


Block Parameters: QR Factorization

QR Factorization (mask)

QR factorization with column pivoting. Q is a unitary matrix, R is an upper triangular matrix (U), and E is a permutation vector such that A(:,E) = Q*R.

References

Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

Supported Data Types

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

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

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 5-8 for related information.

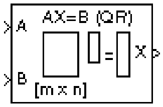
Purpose

Find a minimum-norm-residual solution to the equation A*X*=B

Library

Math Functions / Matrices and Linear Algebra / Linear System Solvers

Description



The QR Solver block solves the linear system A*X*=B, which can be overdetermined, underdetermined, or exactly determined. The system is solved by applying QR factorization to the M-by-N matrix, A, at the A port. The input to the B port is the right side M-by-L matrix, B. A length-M 1-D vector input at either port is treated as an M-by-1 matrix.

The output at the x port is the N-by-L matrix, X. X is always sample based, and is chosen to minimize the sum of the squares of the elements of B-AX. When B is a vector, this solution minimizes the vector 2-norm of the residual (B-AX is the residual). When B is a matrix, this solution minimizes the matrix Frobenius norm of the residual. In this case, the columns of X are the solutions to the L corresponding systems $AX_k=B_k$, where B_k is the kth column of B, and X_k is the kth column of *X*.

X is known as the minimum-norm-residual solution to AX=B. The minimum-norm-residual solution is unique for overdetermined and exactly determined linear systems, but it is not unique for underdetermined linear systems. Thus when the QR Solver is applied to an underdetermined system, the output *X* is chosen such that the number of nonzero entries in X is minimized.

Algorithm

QR factorization factors a column-permuted variant (A_e) of the M-by-N input matrix A as

$$A_e = QR$$

where Q is a M-by-min(M,N) unitary matrix, and R is a min(M,N)-by-N upper-triangular matrix.

The factored matrix is substituted for A_e in

$$A_eX = B_e$$

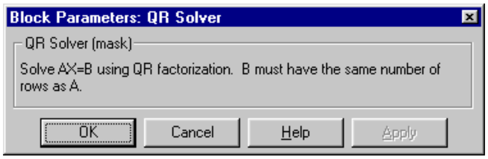
and

$$QRX = B_e$$

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



Block Parameters: QR Solver

QR Solver (mask)

Solve AX=B using QR factorization. B must have the same number of rows as A.

Supported Data Types

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

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

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 5-6 for related information.

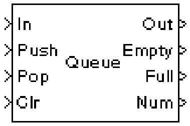
Purpose

Store inputs in a FIFO register

Library

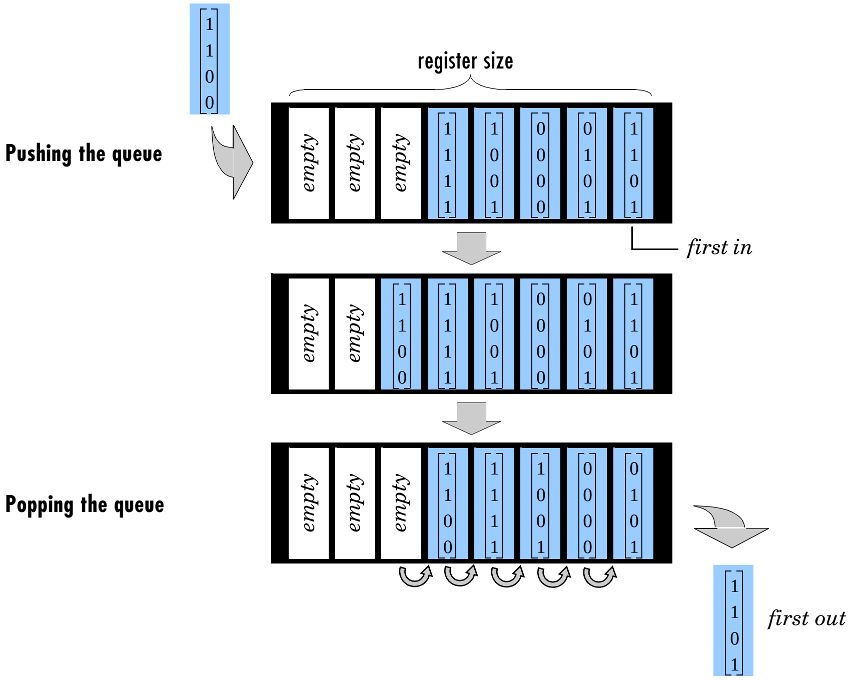
Signal Management / Buffers

Description



The Queue block stores a sequence of input samples in a first in, first out (FIFO) register. The register capacity is set by the **Register size** parameter, and inputs can be scalars, vectors, or matrices.

The block *pushes* the input at the `In` port onto the end of the queue when a trigger event is received at the `Push` port. When a trigger event is received at the `Pop` port, the block *pops* the first element off the queue and holds the `Out` port at that value. The first input to be pushed onto the queue is always the first to be popped off.



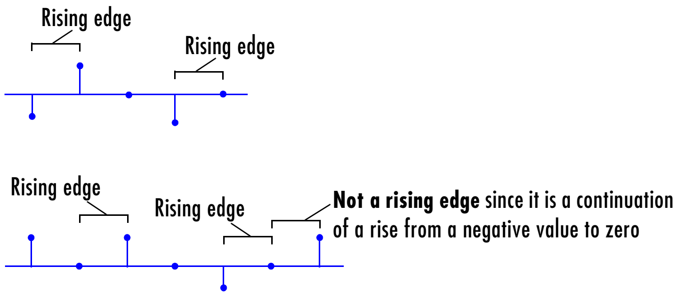
A trigger event at the optional `Clr` port (enabled by the **Clear input** check box) empties the queue contents. If **Clear output port on reset** is selected, then a trigger event at the `Clr` port empties the queue *and* sets the value at the `Out` port to zero. This setting also applies when a disabled 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:

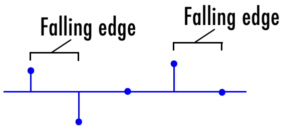
1. `Clr`
2. `Push`
3. `Pop`

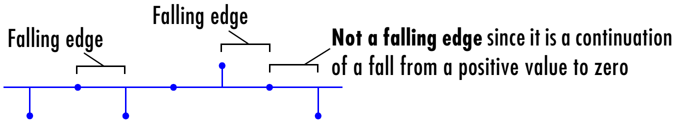
The rate of the trigger signal must be a positive integer multiple of the rate of the data signal input. The triggering event for the `Push`, `Pop`, and `Clr` ports is specified by the **Trigger type** pop-up menu, and can be one of the following.

- `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)


Rising edge
Rising edge
Rising edge
Rising edge
Not a rising edge since it is a continuation of a rise from a negative value to zero

- `Falling edge` — Triggers execution of the block when the trigger input does one of the following:
  - Falls from a positive value to a negative value or zero
  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero (see the following figure)





- `Either edge` — Triggers execution of the block when the trigger input is a `Rising edge` or `Falling edge` (as described above).
- `Non-zero sample` — Triggers execution of the block at each sample time that the trigger input is not zero.

Note When running simulations in the Simulink `MultiTasking` mode, sample-based trigger signals have a one-sample latency, and frame-based trigger signals have one frame of latency. Thus, there is a one-sample or one-frame delay between the time the block detects a trigger event, and when it applies the trigger. For more information on latency and the Simulink tasking modes, see “Excess Algorithmic Delay (Tasking Latency)” on page 2-98 and the topic called 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

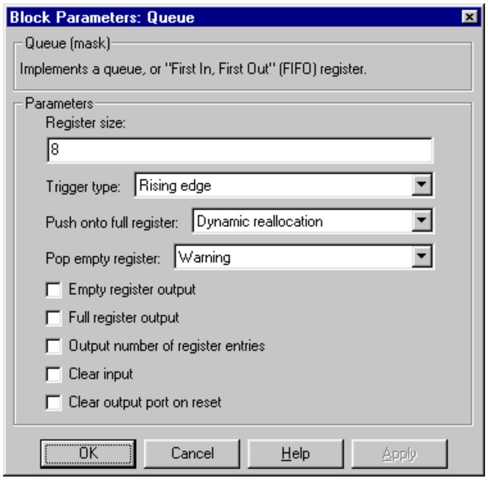
In	Push	Pop	Clr	Queue	Out	Empty	Full	Num
10	1	0	0	*top* □ □ □ 10 *bottom*	5	0	0	1
11	0	0	0	*top* □ □ 11 10 *bottom*	5	0	0	2
12	1	0	1	*top* □ □ □ 12 *bottom*	0	0	0	1

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 the operation of the Queue block.

Dialog Box



Block Parameters: Queue

Queue (mask)

Implements a queue, or "First In, First Out" (FIFO) register.

Parameters

Register size:

8

Trigger type: Rising edge

Push onto full register: Dynamic reallocation

Pop empty register: Warning

☐ Empty register output

☐ Full register output

☐ Output number of register entries

☐ Clear input

☐ Clear output port on reset

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.

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-7. To learn how to disable Boolean output support, see “Steps to Disabling Boolean Support” on page A-8.
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

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

See Also

Buffer	DSP Blockset
Delay Line	DSP Blockset
Stack	DSP Blockset

Random Source

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" on page 7-616
- "Output Complexity" on page 7-617
- "Output Repeatability" on page 7-618
- "Specifying the Initial Seed" on page 7-619
- "Sample Period" on page 7-620
- "Dialog Box" on page 7-621
- "Supported Data Types" on page 7-625
- "See Also" on page 7-625

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 block scalar expands the other parameter so it is the same length as the vector.

When the **Source type** parameter is set to `Gaussian`, you must also set the **Method** parameter, which determines the method by which the block computes the output, and has the following settings:

- `Ziggurat` — Produces Gaussian random values by using the Ziggurat method, which is the same method used by the MATLAB `randn` function.
- `Sum of uniform values` — Produces Gaussian random values by adding and scaling uniformly distributed random signals. 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 block scalar expands the other parameter so it is the same length as the vector.

Output Complexity

The block's output can be either real or complex, as determined by the `Real` and `Complex` options in the **Complexity** parameter. (These settings control all channels of the output, so real and complex data cannot be combined in the same output.) For complex output with a `Uniform` distribution, the real and imaginary components in each channel are both drawn from the same uniform random distribution, defined by the **Minimum** and **Maximum** parameters for that channel.

For complex output with a `Gaussian` distribution, the real and imaginary components in each channel are drawn from normal distributions with different means. In this case, the **Mean** parameter for each channel should specify a complex value; the real component of the **Mean** parameter specifies the mean of the real components in the channel, while the imaginary component specifies the mean of the imaginary components in the channel. 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` — Outputs the same signal each time you run the simulation. The first time you run the simulation, the block randomly selects an initial seed. The block reuses these same initial seeds every time you rerun the simulation.
- `Specify seed` — Outputs the same signal each time you run the simulation. Every time you run the simulation, the block uses the initial seed(s) specified in the **Initial seed** parameter. Also see the next section, “Specifying the Initial Seed” on page 7-619.

- `Not repeatable` — Does not output the same signal each time you run the simulation. Every time you run the simulation, the block randomly selects an initial seed.

Specifying the Initial Seed

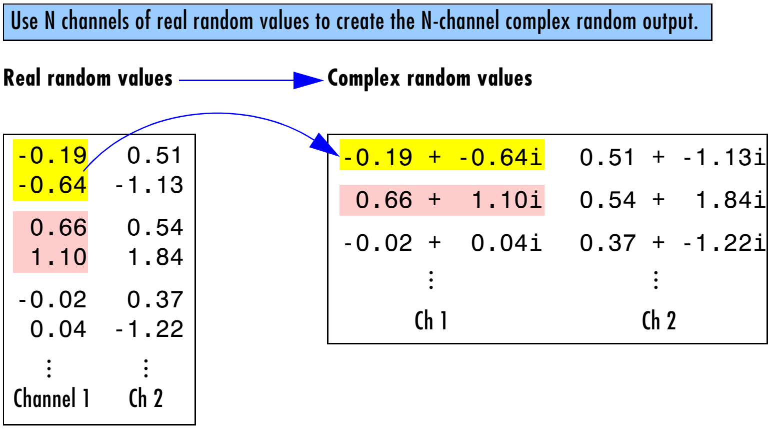
When you set the **Repeatability** parameter to `Specify seed`, you must set the **Initial seed** parameter. The **Initial seed** parameter specifies the initial seed for the pseudorandom number generator. The generator produces an identical sequence of pseudorandom numbers each time it is executed with a particular initial seed.

Specifying Initial Seeds for Real Outputs. To specify the N initial seeds for an N-channel real-valued output (**Complexity** parameter set to `Real`), provide one of the following in the **Initial seed** parameter:

- Length-N vector of initial seeds — Uses each vector element as an initial seed for the corresponding channel in the N-channel output.
- Single scalar — Uses the scalar to generate N random values, which it uses as the seeds for the N-channel output.

Specifying Initial Seeds for Complex Outputs. To specify the initial seeds for an N-channel complex-valued output (**Complexity** parameter set to `Complex`), provide one of the following in the **Initial seed** parameter:

- Length-N vector of initial seeds — Uses each vector element as an initial seed for generating N channels of *real* random values. The block uses pairs of adjacent values in each of these channels as the real and imaginary components of the final output, as illustrated in the following figure.
- Single scalar — Uses the scalar to generate N random values, which it uses as the seeds for generating N channels of *real* random values. The block uses pairs of adjacent values in each of these channels as the real and imaginary components of the final output, as illustrated in the following figure.


Use N channels of real random values to create the N-channel complex random output.
Real random values
Complex random values
-0.19 0.51
-0.64 -1.13
0.66 0.54
1.10 1.84
-0.02 0.37
0.04 -1.22
⋮ ⋮
Channel 1 Ch 2
-0.19 + -0.64i 0.51 + -1.13i
0.66 + 1.10i 0.54 + 1.84i
-0.02 + 0.04i 0.37 + -1.22i
⋮ ⋮
Ch 1 Ch 2

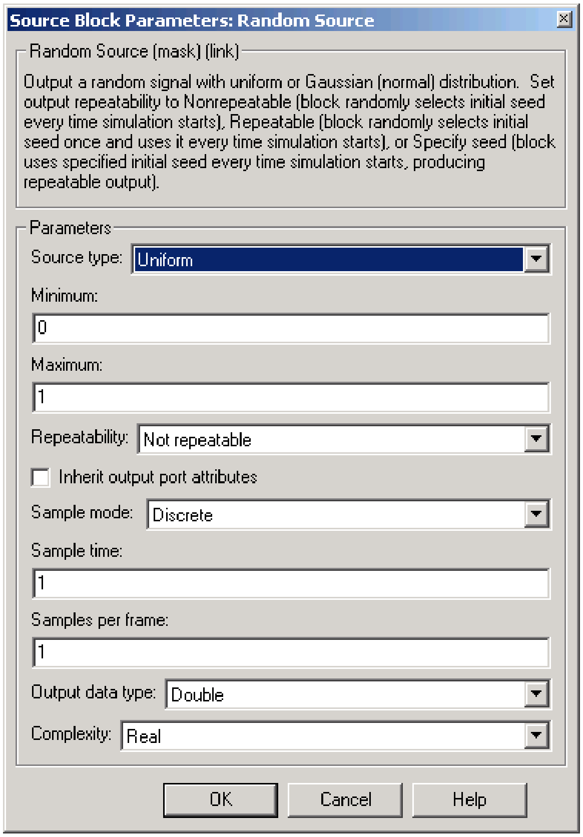
Sample Period

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 Block Parameters: Random Source

Random Source (mask) (link)

Output a random signal with uniform or Gaussian (normal) distribution. Set output repeatability to Nonrepeatable (block randomly selects initial seed every time simulation starts), Repeatable (block randomly selects initial seed once and uses it every time simulation starts), or Specify seed (block uses specified initial seed every time simulation starts, producing repeatable output).

Parameters

Source type: Uniform

Minimum: 0

Maximum: 1

Repeatability: Not repeatable

□ Inherit output port attributes

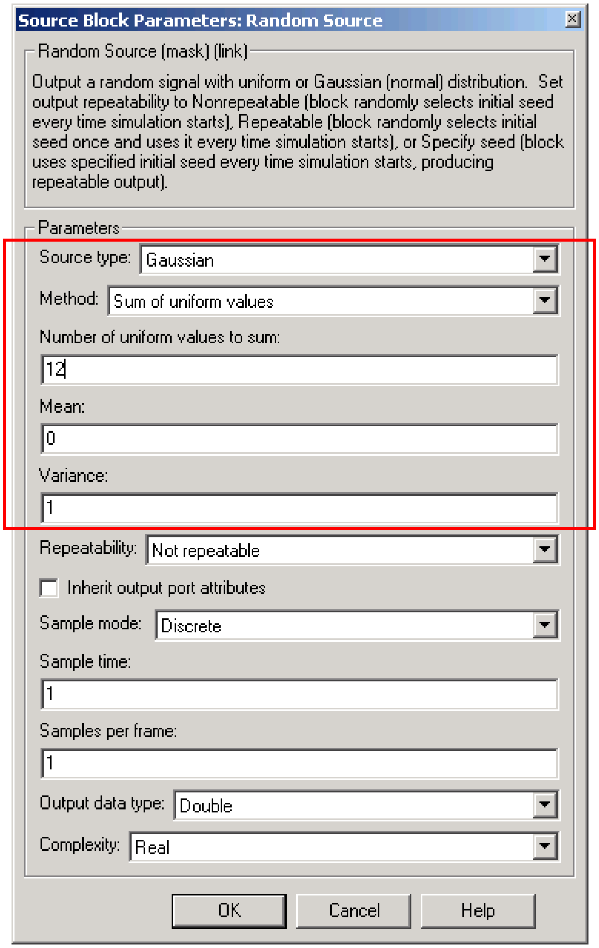
Sample mode: Discrete

Sample time: 1

Samples per frame: 1

Output data type: Double

Complexity: Real



Source Block Parameters: Random Source

Random Source (mask) (link)

Output a random signal with uniform or Gaussian (normal) distribution. Set output repeatability to Nonrepeatable (block randomly selects initial seed every time simulation starts), Repeatable (block randomly selects initial seed once and uses it every time simulation starts), or Specify seed (block uses specified initial seed every time simulation starts, producing repeatable output).

Parameters

Source type: Gaussian

Method: Sum of uniform values

Number of uniform values to sum: 12

Mean: 0

Variance: 1

Repeatability: Not repeatable

□ Inherit output port attributes

Sample mode: Discrete

Sample time: 1

Samples per frame: 1

Output data type: Double

Complexity: Real

Opening this dialog box causes a running simulation to pause. See "Changing Source Block Parameters" in the online Simulink documentation for details.

Source type

The distribution from which to draw the random values, `Uniform` or `Gaussian`. For more information, see "Distribution Type" on page 7-616.

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 7-616.

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 7-616.

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 7-618.

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 7-619.

Inherit output port attributes

When you select this check box, block inherits the sample mode, sample time, output data type, complexity, and signal dimensions of a sample-based signal from a downstream block. When you select this check box, the **Sample mode**, **Sample time**, **Samples per frame**, **Output data type**, and **Complexity** parameters are disabled.

Suppose you want to back propagate a 1-D vector. The output of the Random Source block is a length M sample-based 1-D vector, where length M is inherited from the downstream block. 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.

Suppose you want to back propagate a M-by-N signal. If N>1, your signal has N channels. If N = 1, your signal has M channels. The value of the **Minimum**, **Maximum**, **Mean**, or **Variance** parameter can be a scalar or a vector of length equal to the number of channels. You can specify these parameters as either row or column vectors, except when the signal is a row vector. In this case, the **Minimum**, **Maximum**, **Mean**, or **Variance** parameter must also be specified as a row vector.

Sample mode

The sample mode, `Continuous` or `Discrete`. This parameter is enabled when the **Inherit output port attributes** check box is cleared.

Sample time

The sample period, T_s, of the random output sequence. The output frame period is $M*T_s$. This parameter is enabled when the **Inherit output port attributes** check box is cleared.

Samples per frame

The number of samples, M, in each output frame. This parameter is enabled when the **Inherit output port attributes** check box is cleared.

Output data type

The data type of the output, single-precision or double-precision. This parameter is enabled when the **Inherit output port attributes** check box is cleared.

Output complexity

The complexity of the output, `Real` or `Complex`. This parameter is enabled when the **Inherit output port attributes** check box is cleared.

Supported Data Types

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

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

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 “Creating Signals Using Signal Generator Blocks” on page 2-35 for how to use this and other blocks to generate signals.

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
```

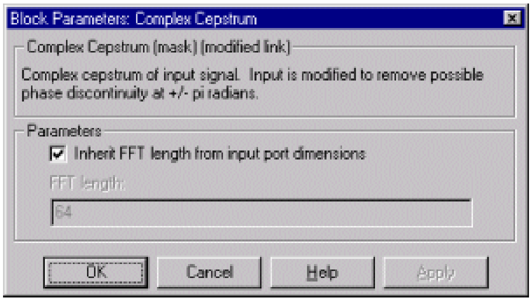
or, more compactly,

```
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 (that is, as an M-by-1 column vector), and the output is a length-M row vector. A 1-D vector input is *always* processed as a single channel, and the output is a 1-D vector.

The output is always sample based, and the output port rate is the same as the input port rate.

Dialog Box



Block Parameters: Complex Cepstrum

Complex Cepstrum (mask) (modified link)

Complex cepstrum of input signal. Input is modified to remove possible phase discontinuity at +/- pi radians.

Parameters

☑ Inherit FFT length from input port dimensions

FFT length: 64

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

See Also

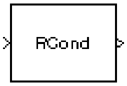
Complex Cepstrum	DSP Blockset
DCT	DSP Blockset
FFT	DSP Blockset
`rceps`	Signal Processing Toolbox

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

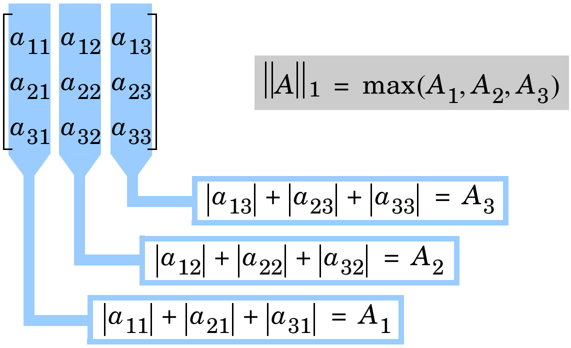
$$y = \frac{1}{\kappa} = \frac{1}{\|A^{-1}\|_1 \|A\|_1}$$

where κ is the condition number ($\kappa \geq 1$), and y is the scalar sample-based output ($0 \leq y < 1$).

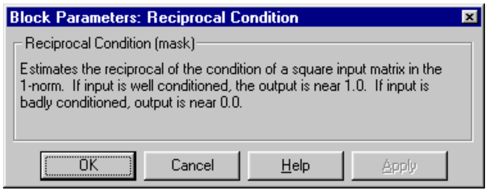
The matrix 1-norm, $\|A\|_1$, is the maximum column-sum in the M-by-M matrix A.

$$\|A\|_1 = \max_{1 \leq j \leq M} \sum_{i=1}^{M} |a_{ij}|$$

For a 3-by-3 matrix:


$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$
$\|A\|_1 = \max(A_1, A_2, A_3)$
$|a_{13}| + |a_{23}| + |a_{33}| = A_3$
$|a_{12}| + |a_{22}| + |a_{32}| = A_2$
$|a_{11}| + |a_{21}| + |a_{31}| = A_1$

Dialog Box



Block Parameters: Reciprocal Condition

Reciprocal Condition (mask)

Estimates the reciprocal of the condition of a square input matrix in the 1-norm. If input is well conditioned, the output is near 1.0. If input is badly conditioned, output is near 0.0.

References

Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

Supported Data Types

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

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

See Also

Matrix 1-Norm	DSP Blockset
Normalization	DSP Blockset
`rcond`	MATLAB

Repeat

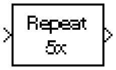
Purpose

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.

This block supports triggered subsystems if, for **Frame-based mode,** you select `Maintain input frame rate`.

Sample-Based Operation

When the input is sample based, the block treats each of the M∗N matrix elements as an independent channel, and upsamples each channel over time. The **Frame-based mode** parameter must be set to `Maintain input frame size`. The output sample rate is L times higher than the input sample rate ($T_{so} = T_{si}/L$), and the input and output sizes are identical.

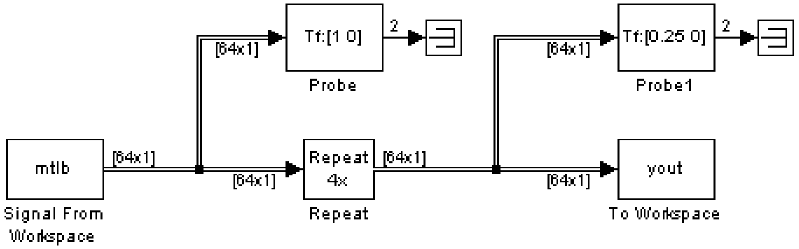
Frame-Based Operation

When the input is frame based, the block treats each of the N input columns as a frame containing M_i sequential time samples from an independent channel. The block upsamples each channel independently by repeating each row of the input matrix L times at the output. The **Frame-based mode** parameter determines how the block adjusts the rate at the output to accommodate the repeated rows. There are two available options:

- `Maintain input frame size`

  The block generates the output at the faster (upsampled) rate by using a proportionally shorter frame *period* at the output port than at the input port. For L repetitions of the input, the output frame period is L times shorter than the input frame period ($T_{fo} = T_{fi}/L$), but the input and output frame sizes are equal.

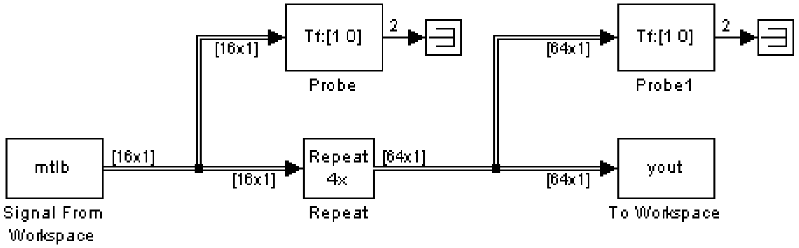
  The model below shows a single-channel input with a frame period of 1 second being upsampled through 4-times repetition to a frame period of 0.25 second. The input and output frame sizes are identical.



- `Maintain input frame rate`

  The block generates the output at the faster (upsampled) rate by using a proportionally larger frame *size* than the input. For L repetitions of the input, the output frame size is L times larger than the input frame size ($M_o = M_i{*}L$), but the input and output frame rates are equal.

  The model below shows a single-channel input of frame size 16 being upsampled through 4-times repetition to a frame size of 64. The input and output frame rates are identical.



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.

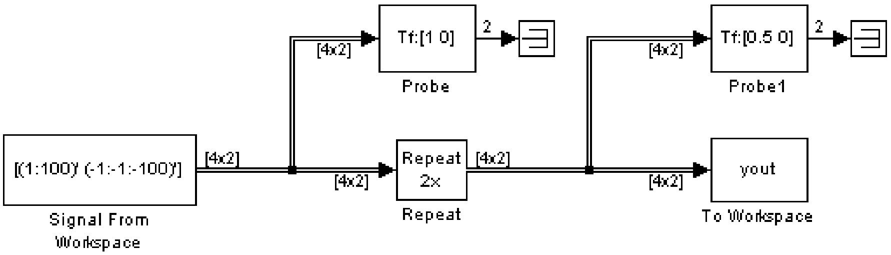
Nonzero Latency. The Repeat block has tasking latency only for multirate operation in the Simulink multitasking mode:

- 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 2-98 and the topic called 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`.

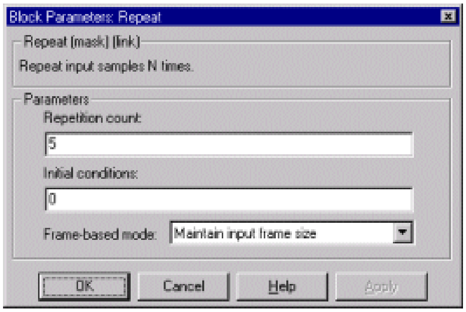
Run the model and look at the output, `yout`. The first few samples of each channel are shown below.

```
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 (that is, sample M_iL+1, where M_i is the input frame size).

Dialog Box



Block Parameters: Repeat

Repeat (mask) (link)

Repeat input samples N times.

Parameters

Repetition count: 5

Initial conditions: 0

Frame-based mode: Maintain input frame size

Repetition count

The integer number of times, L, that the input value is repeated at the output. This is the factor by which the output frame size or sample rate is increased.

Initial conditions

The value with which the block is initialized for cases of nonzero latency; a scalar or matrix.

Frame-based mode

For frame-based operation, the method by which to implement the repetition (upsampling): `Maintain input frame size` (that is, increase the frame rate), or `Maintain input frame rate` (that is, increase the frame size). The **Frame-based mode** parameter must be set to `Maintain input frame size` for sample-base inputs.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point
- Custom data types
- Boolean
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

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

See Also

FIR Interpolation	DSP Blockset
Upsample	DSP Blockset
Zero Pad	DSP Blockset

RLS Adaptive Filter

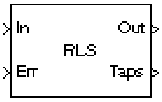
Purpose

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 recursive least squares (RLS) estimate 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, $0.95 \le \lambda \le 1$.

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

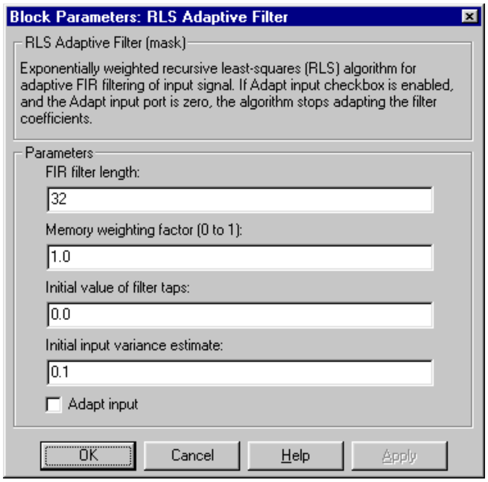
$$I\frac{1}{\hat{\sigma}^2}$$

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



Block Parameters: RLS Adaptive Filter

RLS Adaptive Filter (mask)

Exponentially weighted recursive least-squares (RLS) algorithm for adaptive FIR filtering of input signal. If Adapt input checkbox is enabled, and the Adapt input port is zero, the algorithm stops adapting the filter coefficients.

Parameters

FIR filter length: 32

Memory weighting factor (0 to 1): 1.0

Initial value of filter taps: 0.0

Initial input variance estimate: 0.1

□ Adapt input

FIR filter length

The length of the FIR filter.

Memory weighting factor

The exponential weighting factor, in the range `[0,1]`. A value of `1` specifies an infinite memory. Tunable.

Initial value of filter taps

The initial FIR filter coefficients.

Initial input variance estimate

The initial value of 1/P(n).

Adapt input

Enables the `Adapt` port.

References

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

Supported Data Types

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

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

See Also

Kalman Adaptive Filter	DSP Blockset
LMS Adaptive Filter	DSP Blockset

See “Adaptive Filters” on page 3-46 for related information.

RLS Filter

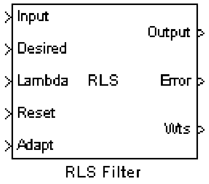
Purpose

Compute the filtered output, filter error, and filter weights for a given input and desired signal using the RLS adaptive filter algorithm

Library

Filtering / Adaptive Filters

Description



The RLS Filter block recursively computes the least squares estimate (RLS) of the FIR filter weights. The block estimates the filter weights, or coefficients, needed to convert the input signal into the desired signal. Connect the signal you want to filter to the Input port. This input can be a sample-based or frame-based signal. Connect the signal you want to model to the Desired port. The desired signal must have the same data type, signal type (sample or frame based), and dimensions as the input signal. The Output port outputs the filtered input signal, which can be sample or frame based. The Error port outputs the result of subtracting the output signal from the desired signal.

The corresponding RLS filter is expressed in matrix form as

$$\begin{aligned}
\mathbf{k}(n) &= \frac{\lambda^{-1}\mathbf{P}(n-1)\mathbf{u}(n)}{1+\lambda^{-1}\mathbf{u}^{H}(n)\mathbf{P}(n-1)\mathbf{u}(n)} \\
y(n) &= \mathbf{w}(n-1)\mathbf{u}(n) \\
e(n) &= d(n)-y(n) \\
\mathbf{w}(n) &= \mathbf{w}(n-1)+\mathbf{k}^{H}(n)e(n) \\
\mathbf{P}(n) &= \lambda^{-1}\mathbf{P}(n-1)-\lambda^{-1}\mathbf{k}(n)\mathbf{u}^{H}(n)\mathbf{P}(n-1)
\end{aligned}$$

where λ^{-1} denotes the reciprocal of the exponential weighting factor. The variables are as follows.

Variable	**Description**
n	The current time index
$\mathbf{u}(n)$	The vector of buffered input samples at step n
$\mathbf{P}(n)$	The inverse correlation matrix at step n
$\mathbf{k}(n)$	The gain vector at step n
$\mathbf{w}(n)$	The vector of filter-tap estimates at step n

Variable	Description
$y(n)$	The filtered output at step n
$e(n)$	The estimation error at step n
$d(n)$	The desired response at step n
λ	The forgetting factor

The implementation of the algorithm in the block is optimized by exploiting the symmetry of the inverse correlation matrix $P(n)$. This decreases the total number of computations by a factor of two.

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

The **Forgetting factor (0 to 1)** parameter corresponds to λ in the equations. It specifies how quickly the filter "forgets" past sample information. Setting λ=1 specifies an infinite memory. Typically, $1 - \frac{1}{2L} < \lambda < 1$, where L is the filter length. You can specify a forgetting factor using the input port, Lambda, or enter a value in the **Forgetting factor (0 to 1)** parameter in the **Block Parameters: RLS Filter** dialog box.

Enter the initial filter weights, $\hat{w}(0)$, as a vector or a scalar for the **Initial value of filter weights** parameter. If you enter a scalar, the block uses the scalar value to create a vector of filter weights. This vector has length equal to the filter length and all of its values are equal to the scalar value.

The initial value of $P(n)$ is

$$\frac{1}{\sigma^2}I$$

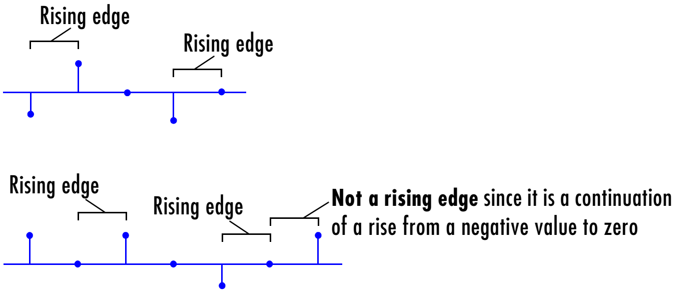
where σ^2 is specified by the **Initial input variance estimate** parameter.

If you select the **Enable/disable adaptation via input port** check box, an Adapt port appears on the block. When the input to this port is nonzero, the block continuously updates the filter weights. When the input to this port is zero, the filter weights remain at their current values.

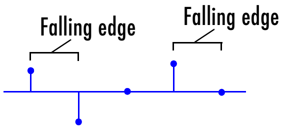
If you want to reset the value of the filter weights to their initial values, use the **Reset input** parameter. The block resets the filter weights whenever a reset event is detected at the Reset port. The reset signal rate must be the same rate as the data signal input.

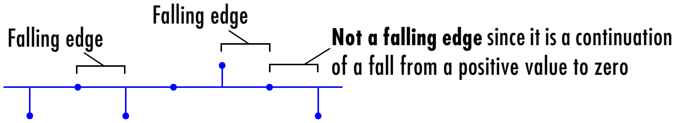
From the **Reset input** list, select `None` to disable the Reset port. To enable the Reset port, select one of the following from the **Reset input** list:

- `Rising edge` — Triggers a reset operation when the Reset input does one of the following:
  - Rises from a negative value to a positive value or zero
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)


Rising edge
Rising edge
Rising edge
Rising edge
Not a rising edge since it is a continuation of a rise from a negative value to zero

- `Falling edge` — Triggers a reset operation when the Reset input does one of the following:
  - Falls from a positive value to a negative value or zero
  - Falls from zero to a negative value, where the fall is not a continuation of a fall from a positive value to zero (see the following figure)




Falling edge
Falling edge
Not a falling edge since it is a continuation of a fall from a positive value to zero

- `Either edge` — Triggers a reset operation when the Reset input is a `Rising edge` or `Falling edge` (as described above)
- `Non-zero sample` — Triggers a reset operation at each sample time that the Reset input is not zero

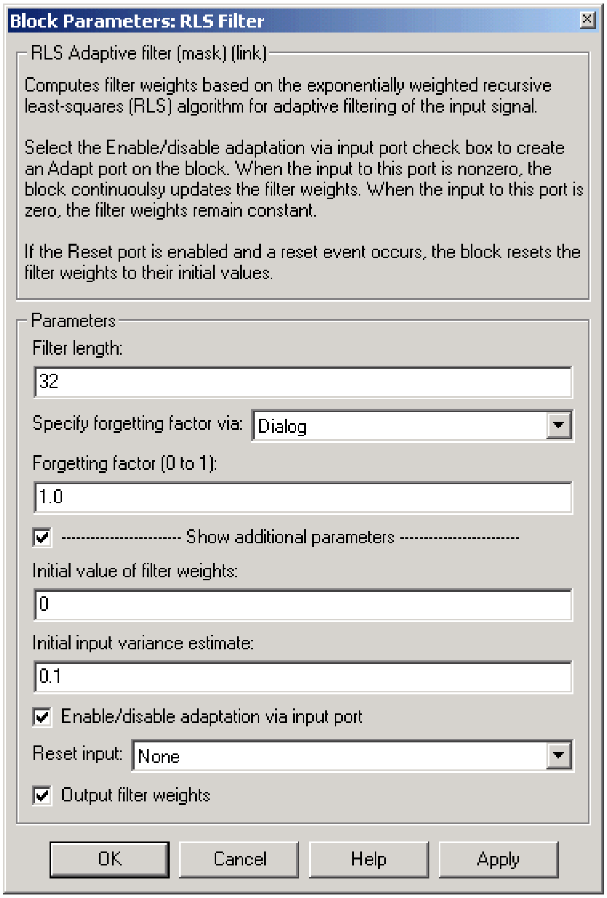
Note When running simulations in the Simulink `MultiTasking` mode, sample-based reset signals have a one-sample latency, and frame-based reset signals have one frame of latency. Thus, there is a one-sample or one-frame delay between the time the block detects a reset event, and when it applies the reset. For more information on latency and the Simulink tasking modes, see “Excess Algorithmic Delay (Tasking Latency)” on page 2-98 and the topic called “The Simulation Parameters Dialog Box” in the Simulink documentation.

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

Example

The `rlsdemo` demo illustrates a noise cancellation system built around the RLS Filter block.

Dialog Box



Block Parameters: RLS Filter

RLS Adaptive filter (mask) (link)

Computes filter weights based on the exponentially weighted recursive least-squares (RLS) algorithm for adaptive filtering of the input signal.

Select the Enable/disable adaptation via input port check box to create an Adapt port on the block. When the input to this port is nonzero, the block continuoulsy updates the filter weights. When the input to this port is zero, the filter weights remain constant.

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

Parameters

Filter length: 32

Specify forgetting factor via: Dialog

Forgetting factor (0 to 1): 1.0

☑ ---------------------- Show additional parameters ----------------------

Initial value of filter weights: 0

Initial input variance estimate: 0.1

☑ Enable/disable adaptation via input port

Reset input: None

☑ Output filter weights

Filter length

Enter the length of the FIR filter weights vector.

Specify forgetting factor via

Select `Dialog` to enter a value for the forgetting factor in the **Block parameters: RLS Filter** dialog box. Select `Input port` to specify the forgetting factor using the Lambda input port.

Forgetting factor (0 to 1)

Enter the exponential weighting factor in the range $0 \le \lambda \le 1$. A value of 1 specifies an infinite memory. Tunable.

Initial value of filter weights

Specify the initial values of the FIR filter weights.

Initial input variance estimate

The initial value of 1/P(n).

Enable/disable adaptation via input port

Select this check box to enable the Adapt input port.

Reset input

Select this check box to enable the Reset input port.

Output filter weights

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

References

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

Supported Data Types

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

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

See Also

Kalman Adaptive Filter	DSP Blockset
LMS Filter	DSP Blockset
Block LMS Filter	DSP Blockset
Fast Block LMS Filter	DSP Blockset

See “Adaptive Filters” on page 3-46 for related information.

RMS

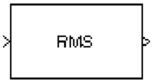
Purpose

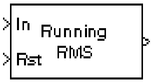
Compute the root-mean-square (RMS) value of an input or sequence of inputs

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 *j*th column is

$$y_j = \sqrt{\frac{\sum_{i=1}^{M} u_{ij}^2}{M}}$$

The frame status of the output is the same as that of the input.

Running Operation

When 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 *j*th column over all inputs since the last reset, up to and including element u_{ij} of the current input.

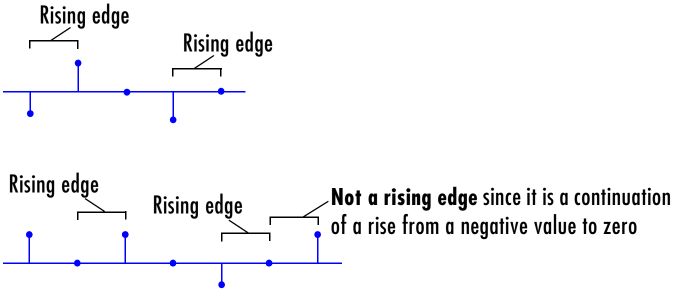
As in basic operation, length-M 1-D vector inputs and *sample-based* length-M row vector inputs are both treated as M-by-1 column vectors.

Resetting the Running RMS. The block resets the running RMS whenever a reset event is detected at the optional `Rst` port. The reset signal rate must be a positive integer multiple of the rate of the data signal input.

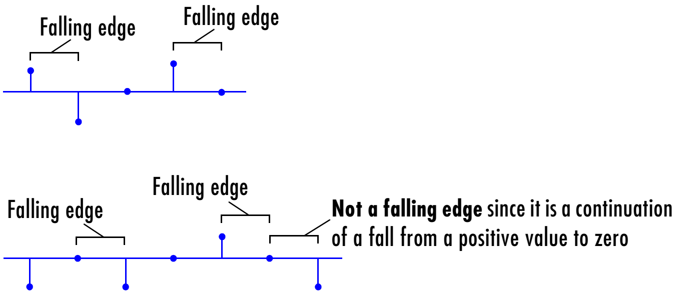
When the block is reset for sample-based inputs, the running RMS for each channel is initialized to the value in the corresponding channel of the current input. For frame-based inputs, the running RMS for each channel is initialized to the earliest value in each channel of the current input.

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)


Rising edge
Rising edge
Rising edge
Rising edge
Not a rising edge since it is a continuation of a rise from a negative value to zero

- `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)

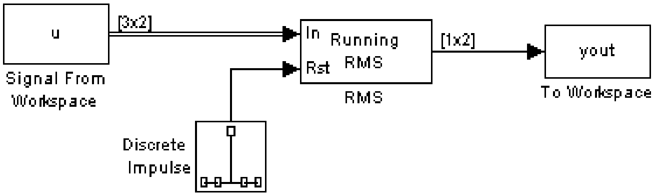

Falling edge
Falling edge
Falling edge
Falling edge
Not a falling edge since it is a continuation of a fall from a positive value to zero

- `Either edge` — Triggers a reset operation when the `Rst` input is a `Rising edge` or `Falling edge` (as described above).
- `Non-zero sample` — Triggers a reset operation at each sample time that the `Rst` input is not zero.

Note When running simulations in the Simulink `MultiTasking` mode, sample-based reset signals have a one-sample latency, and frame-based reset signals have one frame of latency. Thus, there is a one-sample or one-frame delay between the time the block detects a reset event, and when it applies the reset. For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" on page 2-98 and the topic called 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** = Select this check box
- **Reset port** = `Non-zero sample`

The Signal From Workspace block has the following settings:

- **Signal** = `u`
- **Sample time** = `1/3`
- **Samples per frame** = `3`

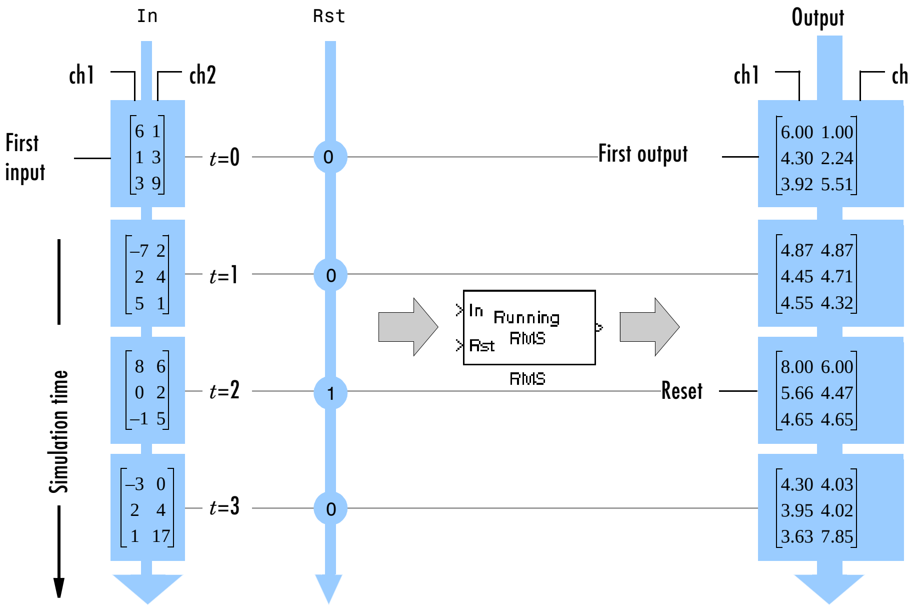
where

```
u = [6 1 3 -7 2 5 8 0 -1 -3 2 1;1 3 9 2 4 1 6 2 5 0 4 17]'
```

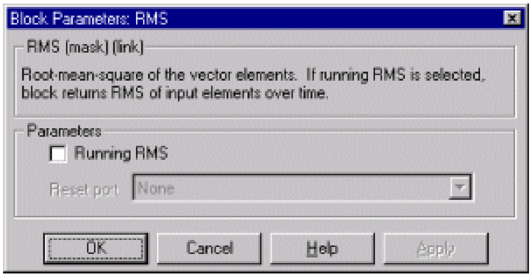
The Discrete Impulse block has the following settings:

- **Delay (samples)** = `2`
- **Sample time** = `1`
- **Samples per frame** = `1`

The block's operation is shown in the figure below.



Dialog Box


Block Parameters: RMS

RMS (mask) (link)

Root-mean-square of the vector elements. If running RMS is selected, block returns RMS of input elements over time.

Parameters

□ Running RMS

Reset port: None

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 7-647.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Boolean — The block accepts Boolean inputs to the Rst port.

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

See Also

Mean	DSP Blockset
Variance	DSP Blockset

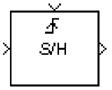
Purpose

Sample and hold an input signal

Library

Signal Operations

Description



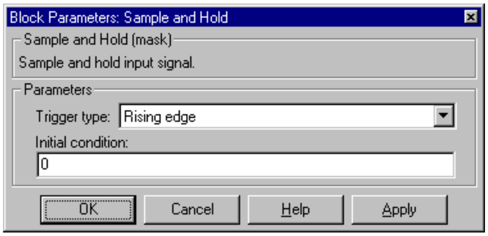
The Sample and Hold block acquires the input at the signal port whenever it receives a trigger event at the trigger port (marked by ⨍). The block then holds the output at the acquired input value until the next triggering event occurs. If the acquired input is frame based, the output is frame based; 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


Block Parameters: Sample and Hold

Sample and Hold (mask)

Sample and hold input signal.

Parameters

Trigger type: Rising edge

Initial condition: 0

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

See Also

Downsample	DSP Blockset
N-Sample Switch	DSP Blockset

Scalar Quantizer

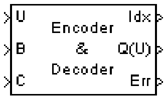
Purpose

Convert an input signal into a set of quantized output values. Convert an input signal into a set of index values. Convert a set of index values into a quantized output signal.

Library

Quantizers

Description



Scalar Quantizer

The Scalar Quantizer block has three modes of operation. In `Encoder` mode, the block maps each input value to a quantization region by comparing the input value to the quantizer boundary points defined in the **Boundary points** parameter. The block outputs the index of the associated region. In `Decoder` mode, the block transforms the input index values into quantized output values, defined in the **Codebook** parameter. In the `Encoder and Decoder` mode, the block performs both the encoding and decoding operations. The block outputs the index values and the quantized output values.

You can select how you want to enter the **Boundary points** and/or **Codebook** values using the **Source of quantizer parameters**. If you select `Specify via dialog`, type the parameters into the block parameters dialog box. Select `Input ports`, and port B and/or C appears on the block. In `Encoder` and `Encoder and decoder` mode, the input to port B is used as the **Boundary points**. In `Decoder` and `Encoder and decoder` mode, the input to port C is used as the **Codebook**.

In `Encoder` and `Encoder and decoder` mode, the **Boundary points** are the values used to break up the input signal into regions. Each region is specified by an index number. If your first boundary point is `-inf` and your last boundary point is `inf`, your quantizer is unbounded. If your first and last boundary point is finite, your quantizer is bounded. If only your first or last boundary point is `-inf` or `inf`, your quantizer is semi-bounded.

For instance, if your input signal ranges from 0 to 11, you can create a bounded quantizer using the following boundary points:

```
[0 0.5 3.7 5.8 6.0 11]
```

The boundary points can have equal or varied spacing. Any input values between 0 and 0.5 would correspond to index 0. Input values between 0.5 and 3.7 would correspond to index 1, and so on.

Suppose you wanted to create an unbounded quantizer with the following boundary points:

```
[-inf 0 2 5.5 7.1 10 inf]
```

If your input signal has values less than 0, these values would be assigned to index 0. If your input signal has values greater than 10, these values would be assigned to index 6.

If an input value is the same as a boundary point, the **Tie-breaking rule** parameter defines the index to which the value is assigned. If you want the input value to be assigned to the lower index value, select `Choose the lower index`. To assign the input value with the higher index, select `Choose the higher index`.

In `Decoder` and `Encoder and decoder` mode, the **Codebook** is a vector of quantized output values that correspond to each index value.

In `Encoder` and `Encoder and decoder` mode, the **Searching method** determines how the appropriate quantizer index is found. Select `Linear` and the Scalar Quantizer block compares the input value to the first region defined by the first two boundary points. If the input value does not fall within this region, the block then compares the input value to the next region. This process continues until the input value is determined to be within a region and is associated with the appropriate index value. The computational cost of this process is of the order P, where P is the number of boundary points.

Select `Binary` for the **Searching method** and the block compares the input value to the middle value of the boundary points vector. If the input value is larger than this boundary point, the block discards the boundary points that are lower than this middle value. The block then compares the input value to the middle boundary point of the new range, defined by the remaining boundary points. This process continues until the input value is associated with the appropriate index value. The computational cost of this process is of the order $\log_2 P$, where P is the number of boundary points. In most cases, the `Binary` option is faster than the `Linear` option.

In `Decoder` mode, the input to this block is a vector of index values, where $0 \le index < N$ and N is the length of the codebook vector. Use the **Action for out of range input** parameter to determine what happens if an input index value is out of this range. If you want any index values less than 0 to be set to 0 and any index values greater than or equal to N to be set to N-1, select `Clip`. If you want to be warned when any index values less than 0 are set to 0 and any index values greater than or equal to N are set to N-1, select `Clip and` warn. If you want the simulation to stop and display an error when the index values are out of range, select Error.

In Encoder and decoder mode, you can select the **Output the quantization error** check box. The quantization error is the difference between the input value and the quantized output value. Select this check box to output the quantization error for each input value from the Err port on this block.

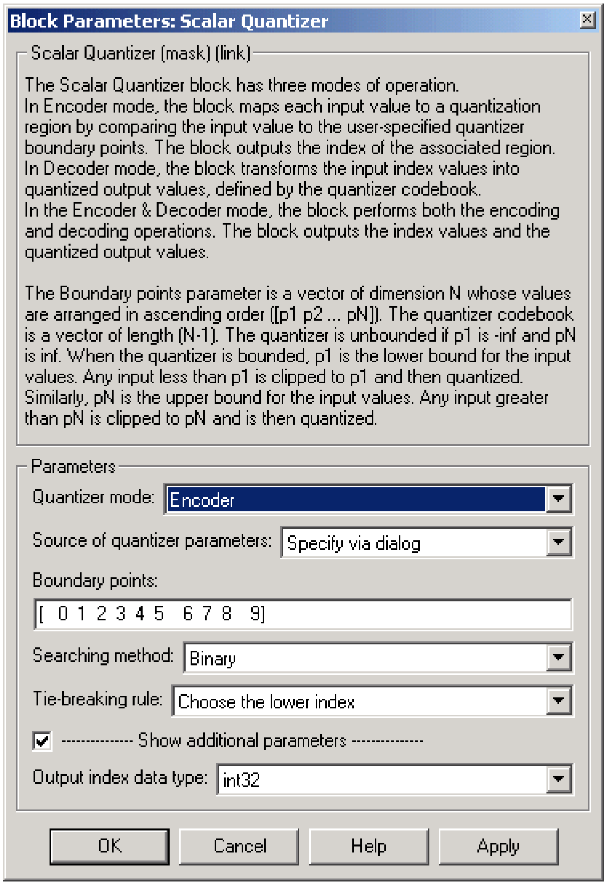
Data Type Support

In Encoder mode, the input data values and the boundary points can be the input to the block at ports U and B. Similarly, in Encoder and decoder mode, the codebook values can also be the input to the block at port C. The data type of the input data values, boundary points, and codebook values can be double, single, uint8, uint16, uint32, int8, int16, or int32. In Decoder mode, the input to the block can be the index values and the codebook values. The data type of the index input to the block at port Idx can be uint8, uint16, uint32, int8, int16, or int32. The data type of the codebook values can be double, single, uint8, uint16, uint32, int8, int16, or int32.

In Encoder mode, the output of the block is the index values. In Encoder and decoder mode, the output can also include the quantized output values and the quantization error. In Encoder and Encoder and decoder mode, use the **Output index data type** parameter to specify the data type of the index output from the block at port Idx. The data type of the index output can be uint8, uint16, uint32, int8, int16, or int32. The data type of the quantized output and the quantization error can be double, single, uint8, uint16, uint32, int8, int16, or int32. In Decoder mode, the output of the block is the quantized output values. Use the **Output data type** parameter to specify the data type of the quantized output values. The data type can be double, single, uint8, uint16, uint32, int8, int16, int32.

Note The input data, codebook values, boundary points, quantization error, and the quantized output values must have the same data type whenever they are present in any of the quantizer modes.

Dialog Box



Block Parameters: Scalar Quantizer

Scalar Quantizer (mask) (link)

The Scalar Quantizer block has three modes of operation.
In Encoder mode, the block maps each input value to a quantization region by comparing the input value to the user-specified quantizer boundary points. The block outputs the index of the associated region.
In Decoder mode, the block transforms the input index values into quantized output values, defined by the quantizer codebook.
In the Encoder & Decoder mode, the block performs both the encoding and decoding operations. The block outputs the index values and the quantized output values.

The Boundary points parameter is a vector of dimension N whose values are arranged in ascending order ([p1 p2 ... pN]). The quantizer codebook is a vector of length (N-1). The quantizer is unbounded if p1 is -inf and pN is inf. When the quantizer is bounded, p1 is the lower bound for the input values. Any input less than p1 is clipped to p1 and then quantized. Similarly, pN is the upper bound for the input values. Any input greater than pN is clipped to pN and is then quantized.

Parameters

Quantizer mode: Encoder

Source of quantizer parameters: Specify via dialog

Boundary points:

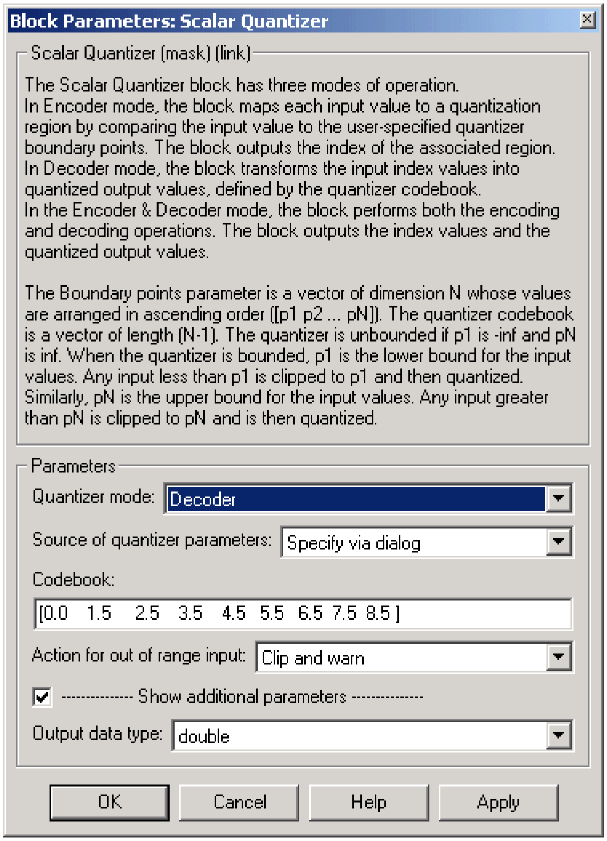
[0 1 2 3 4 5 6 7 8 9]

Searching method: Binary

Tie-breaking rule: Choose the lower index

☑ -------------- Show additional parameters --------------

Output index data type: int32



Scalar Quantizer (mask) (link)

The Scalar Quantizer block has three modes of operation.
In Encoder mode, the block maps each input value to a quantization region by comparing the input value to the user-specified quantizer boundary points. The block outputs the index of the associated region.
In Decoder mode, the block transforms the input index values into quantized output values, defined by the quantizer codebook.
In the Encoder & Decoder mode, the block performs both the encoding and decoding operations. The block outputs the index values and the quantized output values.

The Boundary points parameter is a vector of dimension N whose values are arranged in ascending order ([p1 p2 ... pN]). The quantizer codebook is a vector of length (N-1). The quantizer is unbounded if p1 is -inf and pN is inf. When the quantizer is bounded, p1 is the lower bound for the input values. Any input less than p1 is clipped to p1 and then quantized. Similarly, pN is the upper bound for the input values. Any input greater than pN is clipped to pN and is then quantized.

Parameters

Quantizer mode: Decoder

Source of quantizer parameters: Specify via dialog

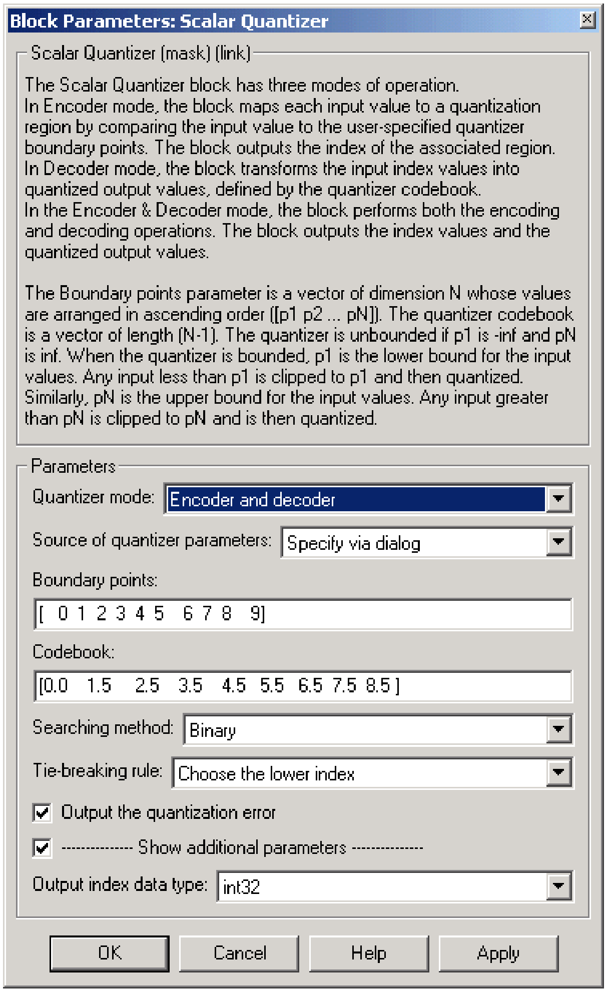
Codebook:

[0.0   1.5   2.5   3.5   4.5 5.5 6.5 7.5 8.5]

Action for out of range input: Clip and warn

☑ -------------- Show additional parameters --------------

Output data type: double



Scalar Quantizer (mask) (link)

The Scalar Quantizer block has three modes of operation.
In Encoder mode, the block maps each input value to a quantization region by comparing the input value to the user-specified quantizer boundary points. The block outputs the index of the associated region.
In Decoder mode, the block transforms the input index values into quantized output values, defined by the quantizer codebook.
In the Encoder & Decoder mode, the block performs both the encoding and decoding operations. The block outputs the index values and the quantized output values.

The Boundary points parameter is a vector of dimension N whose values are arranged in ascending order ([p1 p2 ... pN]). The quantizer codebook is a vector of length (N-1). The quantizer is unbounded if p1 is -inf and pN is inf. When the quantizer is bounded, p1 is the lower bound for the input values. Any input less than p1 is clipped to p1 and then quantized. Similarly, pN is the upper bound for the input values. Any input greater than pN is clipped to pN and is then quantized.

Parameters

Quantizer mode: Encoder and decoder

Source of quantizer parameters: Specify via dialog

Boundary points:

[0 1 2 3 4 5 6 7 8 9]

Codebook:

[0.0 1.5 2.5 3.5 4.5 5.5 6.5 7.5 8.5]

Searching method: Binary

Tie-breaking rule: Choose the lower index

☑ Output the quantization error

☑ -------------- Show additional parameters --------------

Output index data type: int32

Quantizer mode

Specify `Encoder`, `Decoder`, or `Encoder and decoder` as a mode of operation. Nontunable.

Source of quantizer parameters

Choose `Specify via dialog` to type the parameters into the block parameters dialog box. Select `Input ports` to specify the parameters using the block's input ports. In `Encoder` and `Encoder and decoder` mode, input the **Boundary points** using port B. In `Decoder` and `Encoder and decoder` mode, input the **Codebook** values using port C. Nontunable.

Boundary points

Enter a vector of values that represent the boundary points of the quantizer regions. Tunable.

Codebook

Enter a vector of quantized output values that correspond to each index value. Tunable.

Searching method

Select `Linear` and the block finds the region in which the input value is located using a linear search. Select `Binary` and the block finds the region in which the input value is located using a binary search. Nontunable.

Tie-breaking rule

Set this parameter to determine the behavior of the block when the input value is the same as the boundary point. If you select `Choose the lower index`, the input value is assigned to lower index value. If you select `Choose the higher index`, the value is assigned to the higher index. Nontunable.

Action for out of range input

Choose the block's behavior if an input index value is out of range, where $0 \le index < N$ and N is the length of the codebook vector. Select `Clip`, if you want any index values less than 0 to be set to 0 and any index values greater than or equal to N to be set to N-1. Select `Clip and warn`, if you want to be warned when any index values less than 0 are set to 0 and any index values greater than or equal to N are set to N-1. Select `Error`, if you want the simulation to stop and display an error when the index values are out of range. Nontunable.

Output the quantization error

In `Encoder and decoder` mode, select this check box to output the quantization error from the Err port on this block. Nontunable.

Output index data type

In `Encoder` and `Encoder and decoder` mode, specify the data type of the index output from the block at port Idx. The data type can be `uint8`, `uint16`, `uint32`, `int8`, `int16`, or `int32`. This parameter becomes visible when you select the **Show additional parameters** check box. Nontunable.

Output data type

In `Decoder` mode, specify the data type of the quantized output. The data type can be `uint8`, `uint16`, `uint32`, `int8`, `int16`, `int32`, `single`, or `double`. This parameter becomes visible when you select `Specify via dialog` for the **Source of quantizer parameters** and you select the **Show additional parameters** check box. Nontunable.

References

Gersho, A. and R. Gray. *Vector Quantization and Signal Compression*. Boston: Kluwer Academic Publishers, 1992.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

For more information on what data types are supported for each quantizer mode, see “Data Type Support” on page 7-655. To learn how to convert your data types to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-2.

See Also

Quantizer	Simulink
Scalar Quantizer Design	DSP Blockset
Uniform Encoder	DSP Blockset
Uniform Decoder	DSP Blockset
`quantize`	Filter Design Toolbox

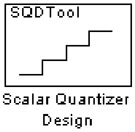
Purpose

Start the Scalar Quantizer Design Tool (SQDTool) to design a scalar quantizer using the Lloyd algorithm

Library

Quantizers

Description



Double-click on the Scalar Quantizer Design block to start SQDTool, a GUI that allows you to design and implement a scalar quantizer. Based on your input values, SQDTool iteratively calculates the codebook values that minimize the mean squared error until the stopping criteria for the design process is satisfied. The block uses the resulting quantizer codebook values and boundary points to implement your scalar quantizer.

For the **Training Set** parameter, enter a set of observations, or samples, of the signal you want to quantize. This data can be any variable defined in the MATLAB workspace including a variable created using a MATLAB function, such as the default value `randn(10000,1)`.

You have two choices for the **Source of initial codebook** parameter. Select `Auto-generate` to have the block choose the values of the initial codebook vector. In this case, the minimum training set value becomes the first codeword, and the maximum training set value becomes the last codeword. Then, the remaining initial codewords are equally spaced between these two values to form a codebook vector of length N, where N is the **Number of levels** parameter. If you select `User defined`, enter the initial codebook values in the **Initial codebook** field. Then, set the **Source of initial boundary points** parameter. You can select `Mid-points` to locate the boundary points at the midpoint between the codewords. To calculate the mid-points, the block internally arranges the initial codebook values in ascending order. You can also choose `User defined` and enter your own boundary points in the **Initial boundary points (unbounded)** field. Only one boundary point can be located between two codewords. If you select `User defined` for the **Source of initial boundary points** parameter, the values you enter in the **Initial codebook** and **Initial boundary points (unbounded)** fields must be arranged in ascending order.

Note This block assumes that you are designing an unbounded quantizer. Therefore, the first and last boundary points are always `-inf` and `inf` regardless of any other boundary point values you might enter.

After you have specified the quantization parameters, the block performs an iterative process to design the optimal scalar quantizer. Each step of the design process involves using the Lloyd algorithm to calculate codebook values and quantizer boundary points. Then, the block calculates the squared quantization error and checks whether the stopping criteria has been satisfied.

The two possible options for the **Stopping criteria** parameter are `Relative threshold` and `Maximum iteration`. If you want the design process to stop when the fractional drop in the squared quantization error is below a certain value, select `Relative threshold`. Then, for **Relative threshold**, type the maximum acceptable fractional drop. If you want the design process to stop after a certain number of iterations, choose `Maximum iteration`. Then, enter the maximum number of iterations you want the block to perform in the **Maximum iteration** field. For **Stopping criteria**, you can also choose `Whichever comes first` and enter a **Relative threshold** and **Maximum iteration** value. The block stops iterating as soon as one of these conditions is satisfied.

With each iteration, the block quantizes the training set values based on the newly calculated codebook values and boundary points. If the training point lies on a boundary point, the algorithm uses the **Tie-breaking rules** parameter to determine which region the value is associated with. If you want the training point to be assigned to the lower indexed region, select `Lower indexed codeword`. To assign the training point with the higher indexed region, select `Higher indexed codeword`.

The **Searching methods** parameter determines how the block compares the training point to the boundary points. Select `Linear search` and SQDTool begins by comparing the training point to the first quantization region defined by `-inf` and the first finite-valued boundary point. If the training point does not fall within this region, the block then compares the training point to the next region. This process continues until the training point is associated with the appropriate region.

Select `Binary search` for the **Searching methods** parameter and the block compares the training point to the middle value of the boundary points vector. If the training point is larger than this boundary point, the block discards the lower boundary points. The block then compares the training point to the middle boundary point of the new range, defined by the remaining boundary points. This process continues until the training point is associated with the appropriate region.

Click **Design and Plot** to design the quantizer with the parameter values specified on the left side of the GUI. The performance curve and the staircase character of the quantizer are updated and displayed in the figures on the right side of the GUI.

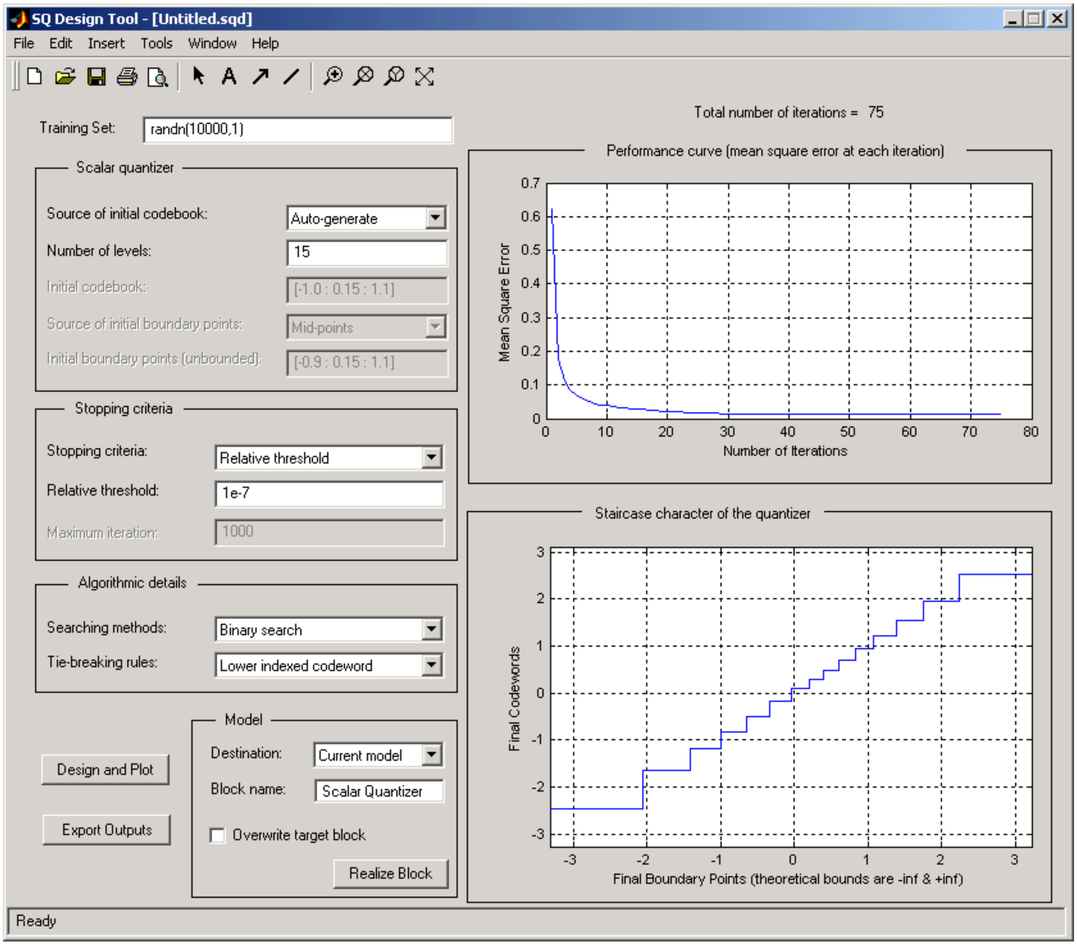
Note You must click **Design and Plot** to apply any changes you make to the parameter values in the SQDTool dialog box.

SQDTool can export parameter values that correspond to the figures displayed in the GUI. Click the **Export Outputs** button, or press **Ctrl+E**, to export the **Final Codebook**, **Final Boundary Points**, and **Error** values to the workspace, a text file, or a MAT-file. The **Error** values represent the mean squared error for each iteration.

In the **Model** section of the GUI, specify the destination of the Scalar Quantizer block that will contain the parameters of your quantizer. For **Destination**, select `Current model` to create a block with your parameters in the model you most recently selected. Type `gcs` in the MATLAB Command Window to display the name of your current model. Select `New model` to create a block in a new model file. In the **Block name** field, enter a name for the block. If you have a Scalar Quantizer block in your destination model with the same name, select the **Overwrite target block** check box to replace the block's parameters with the current parameters. If this check box is not selected, a new Scalar Quantizer block is created in your destination model.

Click **Realize Block**. SQDTool uses the parameters it calculated to set the parameters of the Scalar Quantizer block.

Dialog Box


SQ Design Tool - [Untitled.sqd]
File
Edit
Insert
Tools
Window
Help
Total number of iterations = 75
Performance curve (mean square error at each iteration)
Mean Square Error
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
10
20
30
40
50
60
70
80
Number of Iterations
Staircase character of the quantizer
Final Codewords
3
2
1
0
-1
-2
-3
-3
-2
-1
0
1
2
3
Final Boundary Points (theoretical bounds are -inf & +inf)
Design and Plot
Export Outputs
Realize Block
Ready

Training Set: randn(10000,1)

Scalar quantizer

Source of initial codebook: Auto-generate

Number of levels: 15

Initial codebook: [-1.0 : 0.15 : 1.1]

Source of initial boundary points: Mid-points

Initial boundary points (unbounded): [-0.9 : 0.15 : 1.1]

Stopping criteria

Stopping criteria: Relative threshold

Relative threshold: 1e-7

Maximum iteration: 1000

Algorithmic details

Searching methods: Binary search

Tie-breaking rules: Lower indexed codeword

Model

Destination: Current model

Block name: Scalar Quantizer

□ Overwrite target block

Training Set

Enter the samples of the signal you would like to quantize. This data set can be a MATLAB function or a variable defined in the MATLAB workspace. The typical length of this data vector is `1e6`.

Source of initial codebook

Select `Auto-generate` to have the block choose the initial codebook values. Select `User defined` to enter your own initial codebook values.

Number of levels

Enter the length of the codebook vector. For a b-bit quantizer, the length should be $N = 2^b$.

Initial codebook

Enter your initial codebook values. From the **Source of initial codebook** list, select `User defined` in order to activate this parameter.

Source of initial boundary points

Select `Mid-points` to locate the boundary points at the midpoint between the codebook values. Choose `User defined` to enter your own boundary points. From the **Source of initial codebook** list, select `User defined` in order to activate this parameter.

Initial boundary points (unbounded)

Enter your initial boundary points. This block assumes that you are designing an unbounded quantizer. Therefore, the first and last boundary point are `-inf` and `inf`, regardless of any other boundary point values you might enter. From the **Source of initial boundary points** list, select `User defined` in order to activate this parameter.

Stopping criteria

Choose `Relative threshold` to enter the maximum acceptable fractional drop in the squared quantization error. Choose `Maximum iteration to` specify the number of iterations at which to stop`.` Choose `Whichever comes first` and the block stops the iteration process as soon as the relative threshold or maximum iteration value is attained.

Relative threshold

Type the value that is the maximum acceptable fractional drop in the squared quantization error.

Maximum iteration

Enter the maximum number of iterations you want the block to perform. From the **Stopping criteria** list, select `Maximum iteration` in order to activate this parameter.

Searching methods

Choose `Linear search` to use a linear search method when comparing the training points to the boundary points. Choose `Binary search to` use a binary search method when comparing the training points to the boundary points.

Tie-breaking rules

If a training point lies on a boundary point, choose `Lower indexed codeword` to assign the training point to the lower indexed quantization region. Choose `Higher indexed codeword` to assign the training point to the higher indexed region.

Design and Plot

Click this button to display the performance curve and the staircase character of the quantizer in the figures on the right side of the GUI. These plots are based on the current parameter settings.

You must click **Design and Plot** to apply any changes you make to the parameter values in the SQDTool GUI.

Export Outputs

Click this button, or press **Ctrl+E**, to export the **Final Codebook**, **Final Boundary Points**, and **Error** values to the workspace, a text file, or a MAT-file.

Destination

Choose `Current model` to create a Scalar Quantizer block in the model you most recently selected. Type `gcs` in the MATLAB Command Window to display the name of your current model. Choose `New model` to create a block in a new model file.

Block name

Enter a name for the block.

Overwrite target block

If this check box is not selected, a new Scalar Quantizer block is created in the destination model. If this check box is selected and a Scalar Quantizer block with the same block name exists in the destination model, the parameters of this block are overwritten by new parameters.

Realize Block

Click this button and SQDTool uses the parameters it calculated to set the parameters of the Scalar Quantizer block.

References

Gersho, A. and R. Gray. *Vector Quantization and Signal Compression*. Boston: Kluwer Academic Publishers, 1992.

Supported Data Types

- Double-precision floating point

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

See Also

Quantizer	Simulink
Scalar Quantizer	DSP Blockset
Uniform Encoder	DSP Blockset
Uniform Decoder	DSP Blockset

Short-Time FFT

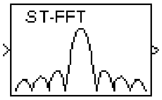
Purpose

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.

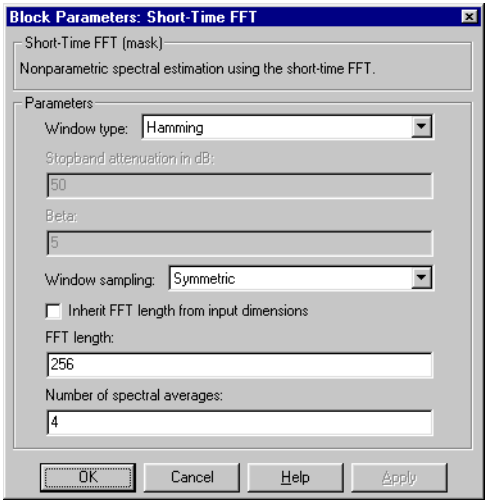
The **Number of spectral averages** specifies the number of spectra to average. Setting this parameter to `1` effectively disables averaging.

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 Short-time FFT block with several other spectral estimation methods.

Dialog Box



Block Parameters: Short-Time FFT

Short-Time FFT (mask)

Nonparametric spectral estimation using the short-time FFT.

Parameters

Window type: Hamming

Stopband attenuation in dB: 50

Beta: 5

Window sampling: Symmetric

□ Inherit FFT length from input dimensions

FFT length: 256

Number of spectral averages: 4

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 your data types to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-2.

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 5-5 for related information.

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 ($M\neq1$), 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 (that is, no linear interpolation takes place). Additionally, the initial signal values are always produced immediately at *t*=0.

When the block has output all of the available signal samples, it can start again at the beginning of the signal, or simply repeat the final value or generate zeros until the end of the simulation. (The block does not extrapolate the imported signal beyond the last sample.) The **Form output after final data value by** parameter controls this behavior:

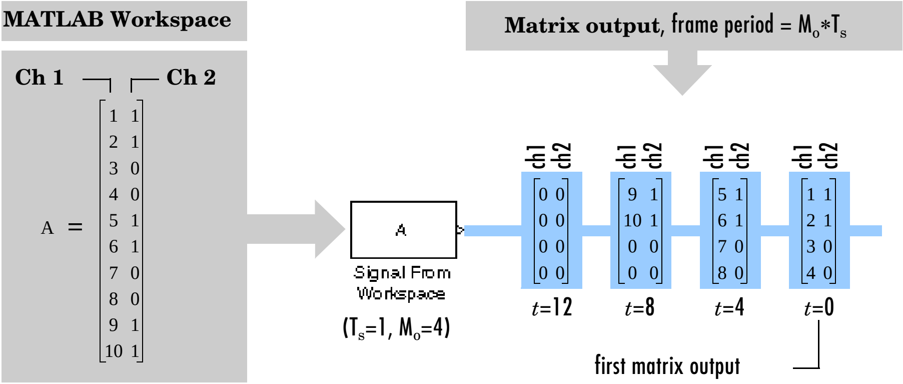
- If `Setting To Zero` is specified, the block generates zero-valued outputs for the duration of the simulation after generating the last frame of the signal.
- If `Holding Final Value` is specified, the block repeats the final sample for the duration of the simulation after generating the last frame 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

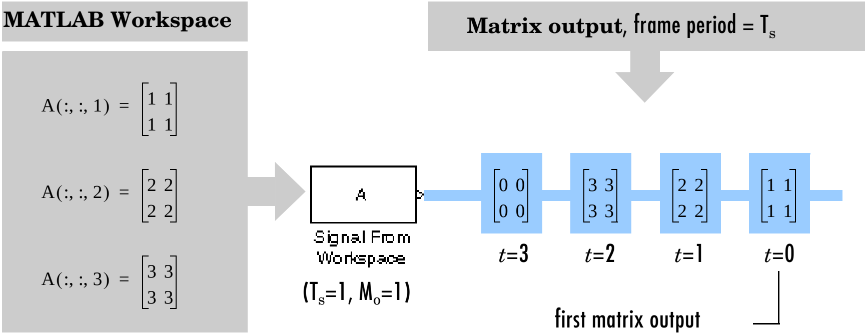
Example 1

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.



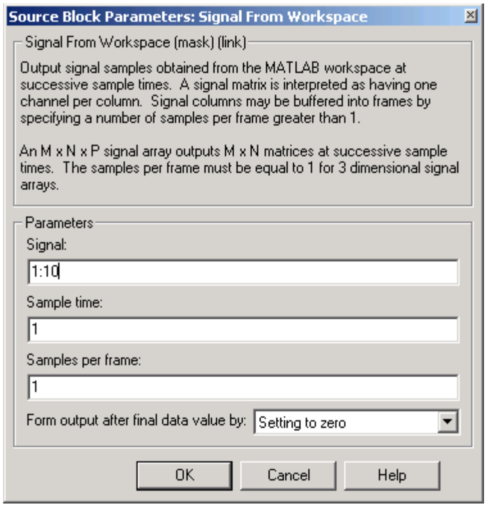
Example 2

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.


MATLAB Workspace
$A(:,:,1) = \begin{bmatrix}1 & 1\\1 & 1\end{bmatrix}$
$A(:,:,2) = \begin{bmatrix}2 & 2\\2 & 2\end{bmatrix}$
$A(:,:,3) = \begin{bmatrix}3 & 3\\3 & 3\end{bmatrix}$
A
Signal From Workspace
$(T_s=1, M_o=1)$
Matrix output, frame period = T_s
$\begin{bmatrix}0 & 0\\0 & 0\end{bmatrix}$ $t=3$
$\begin{bmatrix}3 & 3\\3 & 3\end{bmatrix}$ $t=2$
$\begin{bmatrix}2 & 2\\2 & 2\end{bmatrix}$ $t=1$
$\begin{bmatrix}1 & 1\\1 & 1\end{bmatrix}$ $t=0$
first matrix output

The **Samples per frame** parameter is set to 1 for 3-D input.

Dialog Box


Source Block Parameters: Signal From Workspace

Signal From Workspace (mask) (link)

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

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`).

Supported Data Types

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

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

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 2-2
- "Multichannel Signals" on page 2-10
- "Benefits of Frame-Based Processing" on page 2-13
- "Creating Signals Using the Signal From Workspace Block" on page 2-37
- "Importing Signals" on page 2-69

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 7-675
- "Output Dimension Summary" on page 7-676
- "Matching the Outputs of Signal To Workspace and To Workspace Blocks" on page 7-677
- "Examples" on page 7-678

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 *d*th 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 7-679.)
- `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 7-679.)

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 7-678.)

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.

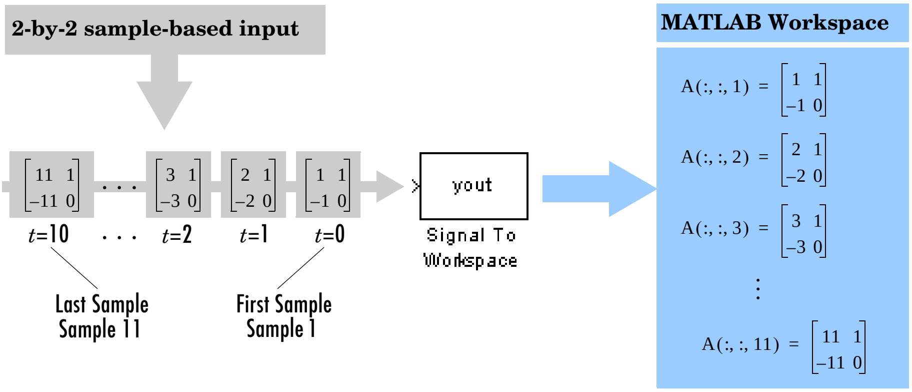
Matching the Outputs of Signal To Workspace and To Workspace Blocks

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 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.


2-by-2 sample-based input
$\begin{bmatrix} 11 & 1 \\ -11 & 0 \end{bmatrix}$ . . . $\begin{bmatrix} 3 & 1 \\ -3 & 0 \end{bmatrix}$ $\begin{bmatrix} 2 & 1 \\ -2 & 0 \end{bmatrix}$ $\begin{bmatrix} 1 & 1 \\ -1 & 0 \end{bmatrix}$
t=10 . . . t=2 t=1 t=0
Last Sample
Sample 11
First Sample
Sample 1
yout
Signal To
Workspace
MATLAB Workspace
$A(:,:,1) = \begin{bmatrix} 1 & 1 \\ -1 & 0 \end{bmatrix}$
$A(:,:,2) = \begin{bmatrix} 2 & 1 \\ -2 & 0 \end{bmatrix}$
$A(:,:,3) = \begin{bmatrix} 3 & 1 \\ -3 & 0 \end{bmatrix}$
⋮
$A(:,:,11) = \begin{bmatrix} 11 & 1 \\ -11 & 0 \end{bmatrix}$

The block settings are as follows.

Signal To Workspace Block Parameters

Variable name	`yout`
Limit data points to last	`inf`
Decimation	`1`
Frames	ignored since block input is not frame based

Simulation Parameters Dialog Parameters

Start time	`0`
Stop time	`10`

Signal From Workspace Parameters (provides Signal To Workspace input)

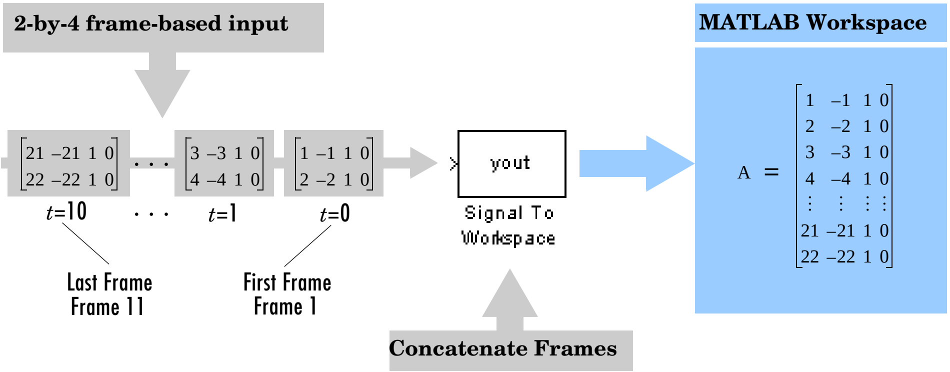
Signal	`input1` (defined below)
Sample time	`1`
Samples per frame	`1`
Form output after final data value by	`Setting to zero`

```
input1 = cat(3, [1 1; -1 0], [2 1; -2 0],...,[11 1; -11 0])
```

Example 2: Frame-Based Inputs. In the following model, the input to the Signal To Workspace block is a 2-by-4 frame-based matrix signal with a frame period of 1 (generated by a Signal From Workspace block). The block logs 11 frames (two samples per frame) by the end of the simulation. The frames are concatenated to create a 22-by-4 matrix, A, in the MATLAB workspace.

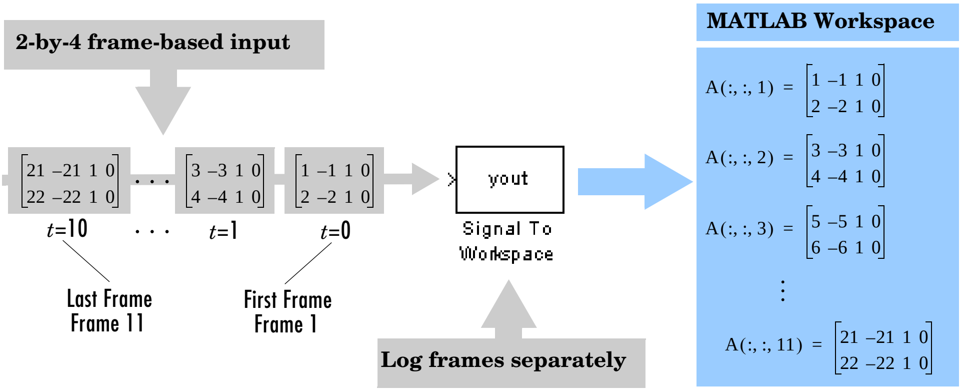
The block settings for the following model are similar to the settings used in Example 1, except **Frames** is set to `Concatenate frames (2-D array)` and the Signal From Workspace parameter, **Signal**, is set to `input2`, where

```
input2 = [1 -1 1 0; 2 -2 1 0; 3 -3 1 0;...; 22 -22 1 0]
```

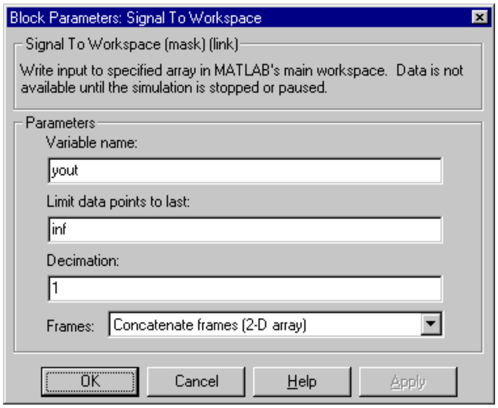


2-by-4 frame-based input
MATLAB Workspace
$\begin{bmatrix}21 & -21 & 1 & 0\\22 & -22 & 1 & 0\end{bmatrix}$
. . .
$\begin{bmatrix}3 & -3 & 1 & 0\\4 & -4 & 1 & 0\end{bmatrix}$
$\begin{bmatrix}1 & -1 & 1 & 0\\2 & -2 & 1 & 0\end{bmatrix}$
yout
Signal To Workspace
$A = \begin{bmatrix}1 & -1 & 1 & 0\\2 & -2 & 1 & 0\\3 & -3 & 1 & 0\\4 & -4 & 1 & 0\\\vdots & \vdots & \vdots & \vdots\\21 & -21 & 1 & 0\\22 & -22 & 1 & 0\end{bmatrix}$
t=10
. . .
t=1
t=0
Last Frame
Frame 11
First Frame
Frame 1
Concatenate Frames

In the 2-D output, there is no indication of where one frame ends and another begins. By setting **Frames** to `Log frames separately (3-D array)` in this model, you can easily see each frame in the MATLAB workspace, as illustrated in the following model. Each of the 11 frames is logged separately to create a 2-by-4-by-11 array, A, in the MATLAB workspace.



Dialog Box



Block Parameters: Signal To Workspace

Signal To Workspace (mask) (link)

Write input to specified array in MATLAB's main workspace. Data is not available until the simulation is stopped or paused.

Parameters

Variable name: yout

Limit data points to last: inf

Decimation: 1

Frames: Concatenate frames (2-D array)

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 your data types to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-2.

See Also

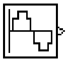
Triggered To Workspace	DSP Blockset
To Workspace	Simulink

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

$$y = A\sin(2\pi ft + \phi)$$

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

$$y = Ae^{j(2\pi ft + \phi)} = A\{\cos(2\pi ft + \phi) + j\sin(2\pi ft + \phi)\}$$

Sections of This Reference Page

- “Generating Multichannel Outputs” on page 7-682
- “Output Sample Time and Samples Per Frame” on page 7-683
- “Sample Mode” on page 7-683
- “Discrete Computational Methods” on page 7-684
- “Examples” on page 7-686
- “Dialog Box” on page 7-687
- “Supported Data Types” on page 7-692
- “See Also” on page 7-692

Generating Multichannel 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 *i*th 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

$$\begin{bmatrix} \sin\{2\pi f_i(t+T_s)+\phi_i\} \\ \cos\{2\pi f_i(t+T_s)+\phi_i\} \end{bmatrix} = \begin{bmatrix} \cos(2\pi f_i T_s) & \sin(2\pi f_i T_s) \\ -\sin(2\pi f_i T_s) & \cos(2\pi f_i T_s) \end{bmatrix} \begin{bmatrix} \sin(2\pi f_i t+\phi_i) \\ \cos(2\pi f_i t+\phi_i) \end{bmatrix}$$

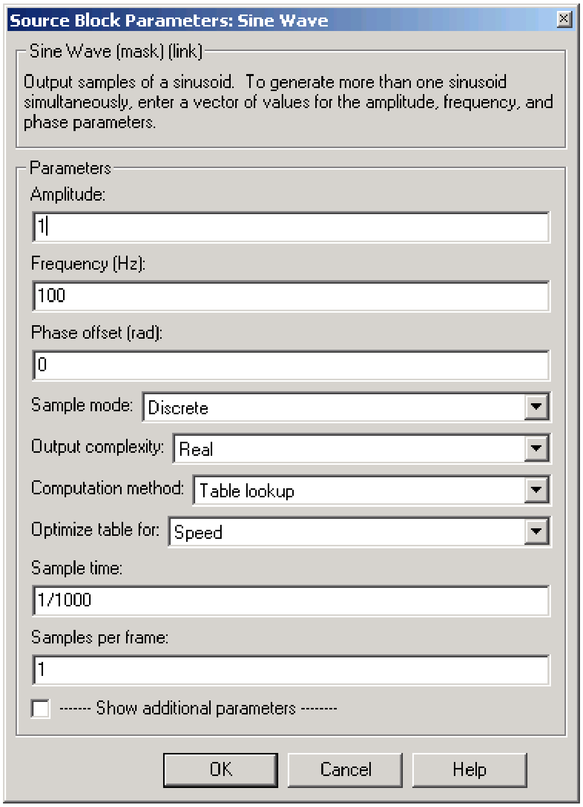
where $\mathrm{T_s}$ is specified by the **Sample time** parameter. Since $\mathrm{T_s}$ is constant, the right-hand matrix is a constant and can be computed once at the start of the simulation. The value of $A_i\sin[2\pi f_i(t+\mathrm{T_s})+\phi_i]$ is then computed from the values of $\sin(2\pi f_i t+\phi_i)$ 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



Source Block Parameters: Sine Wave

Sine Wave (mask) (link)

Output samples of a sinusoid. To generate more than one sinusoid simultaneously, enter a vector of values for the amplitude, frequency, and phase parameters.

Parameters

Amplitude: 1

Frequency (Hz): 100

Phase offset (rad): 0

Sample mode: Discrete

Output complexity: Real

Computation method: Table lookup

Optimize table for: Speed

Sample time: 1/1000

Samples per frame: 1

□ ------ Show additional parameters -------

Opening this dialog box causes a running simulation to pause. See “Changing Source Block Parameters” in the online Simulink documentation for details.

Amplitude

A length-N vector containing the amplitudes of the sine waves in each of N output channels, or a scalar to be applied to all N channels. The vector length must be the same as that specified for the **Frequency** and **Phase** parameters. Tunable (when **Computation method** is *not* set to `Table lookup`); the amplitude values can be altered while a simulation is running, but the vector length must remain the same.

Frequency

A length-N vector containing frequencies, in rad/s, of the sine waves in each of N output channels, or a scalar to be applied to all N channels. The vector length must be the same as that specified for the **Amplitude** and **Phase** parameters. You can specify positive, zero, or negative frequencies. Tunable (when **Computation method** is *not* set to `Table lookup`); the frequency values can be altered while a simulation is running, but the vector length must remain the same. This parameter is not tunable in the Simulink external mode when using the differential method.

Phase offset

A length-N vector containing the phase offsets, in radians, of the sine waves in each of N output channels, or a scalar to be applied to all N channels. The vector length must be the same as that specified for the **Amplitude** and **Frequency** parameters. This parameter is tunable when **Computation method** is *not* set to `Table lookup`; the phase values can be altered while a simulation is running, but the vector length must remain the same. This parameter is not tunable in the Simulink external mode when using the differential method.

Sample mode

The block's sampling behavior, `Continuous` or `Discrete`. This parameter is not tunable.

Output complexity

The type of waveform to generate: `Real` specifies a real sine wave, `Complex` specifies a complex exponential. This parameter is not tunable.

Computation method

The method by which discrete-time sinusoids are generated: **Trigonometric fcn**, **Table lookup**, or **Differential**. This parameter is not tunable. This parameter is disabled when `Continuous` is selected from the **Sample mode** parameter. For details, see "Discrete Computational Methods" on page 7-684.

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

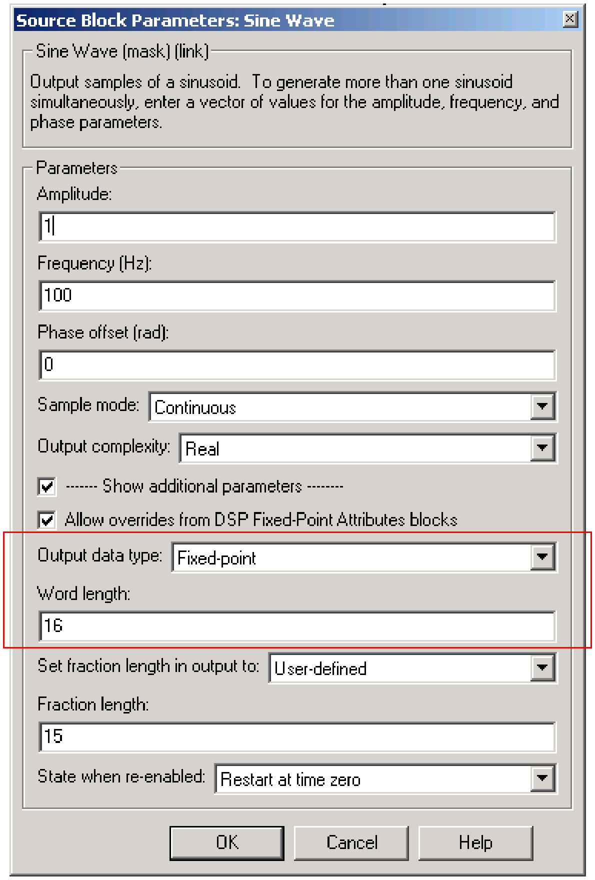
The period with which the sine wave is sampled, T_s. The block's output frame period is $M*T_s$, where M is specified by the **Samples per frame** parameter. This parameter is disabled when `Continuous` is selected from the **Sample mode** parameter. This parameter is not tunable.

Samples per frame

The number of consecutive samples from each sinusoid to buffer into the output frame, M. This parameter is disabled when `Continuous` is selected from the **Sample mode** parameter. Nontunable.

Show additional parameters

If selected, additional parameters specific to implementation of the block become visible as shown.



Source Block Parameters: Sine Wave

Sine Wave (mask) (link)

Output samples of a sinusoid. To generate more than one sinusoid simultaneously, enter a vector of values for the amplitude, frequency, and phase parameters.

Parameters

Amplitude: 1

Frequency (Hz): 100

Phase offset (rad): 0

Sample mode: Continuous

Output complexity: Real

☑ ------ Show additional parameters -------

☑ Allow overrides from DSP Fixed-Point Attributes blocks

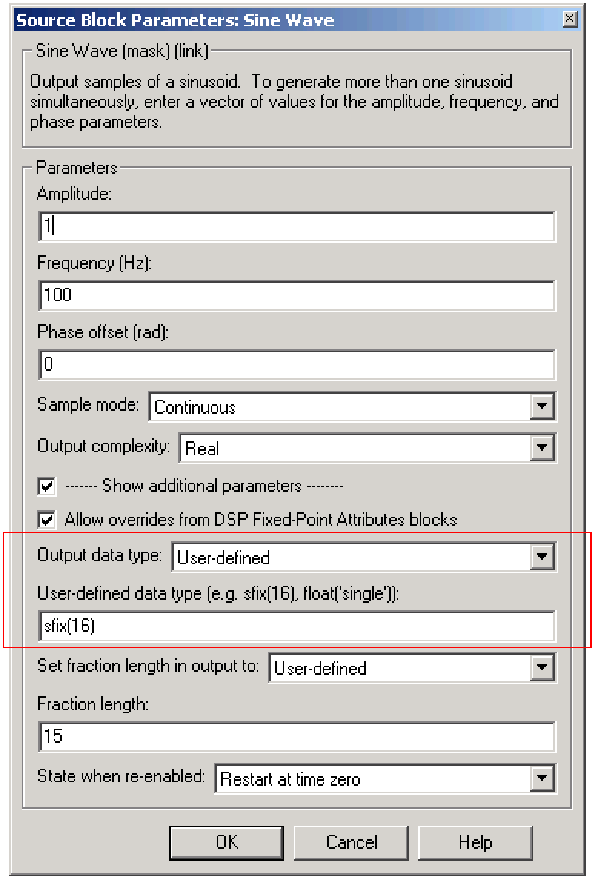
Output data type: Fixed-point

Word length: 16

Set fraction length in output to: User-defined

Fraction length: 15

State when re-enabled: Restart at time zero



Source Block Parameters: Sine Wave

Sine Wave (mask) (link)

Output samples of a sinusoid. To generate more than one sinusoid simultaneously, enter a vector of values for the amplitude, frequency, and phase parameters.

Parameters

Amplitude: 1

Frequency (Hz): 100

Phase offset (rad): 0

Sample mode: Continuous

Output complexity: Real

☑ ------ Show additional parameters -------

☑ Allow overrides from DSP Fixed-Point Attributes blocks

Output data type: User-defined

User-defined data type (e.g. sfix(16), float('single')): sfix(16)

Set fraction length in output to: User-defined

Fraction length: 15

State when re-enabled: Restart at time zero

Allow overrides from DSP Fixed-Point Attributes blocks

If you select this parameter, and if the **Output data type parameter** is set to `Fixed-point`, fixed-point data types for this block may be set by DSP Fixed-Point Attributes blocks in your model. If this parameter is unselected, the data types are always set by the parameters in the block mask.

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 fraction length in output to**, and **Fraction length** parameters.

- Choose `User-defined` to specify the output data type and scaling in the **User-defined data type**, **Set fraction length in output to**, and **Fraction length** parameters.
- Choose `Inherit via back propagation` to set the output data type and scaling to match the next block downstream.

Word length

Specify the word length, in bits, of the fixed-point output data type. This parameter is only visible 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 fraction length in output to

Specify the scaling of the fixed-point output by either of the following two methods:

- Choose `Best precision` to have the output scaling automatically set such that the output signal has the best possible precision.
- Choose `User-defined` to specify the output scaling in the **Fraction length** parameter.

This parameter is only visible if `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.

Fraction length

For fixed-point output data types, specify the number of fractional bits, or bits to the right of the binary point. This parameter is only visible if `Fixed-point` or `User-defined` is selected for the **Output data type** parameter, and if `User-defined` is selected for the **Set fraction length 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 your data types to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-2.

See Also

Chirp	DSP Blockset
Complex Exponential	DSP Blockset
Signal From Workspace	DSP Blockset
Signal Generator	Simulink
Sine Wave	Simulink
`sin`	MATLAB

Also see “Creating Signals Using Signal Generator Blocks” on page 2-35 for how to use this and other blocks to generate signals.

Singular Value Decomposition

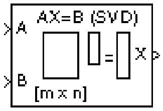
Purpose

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.

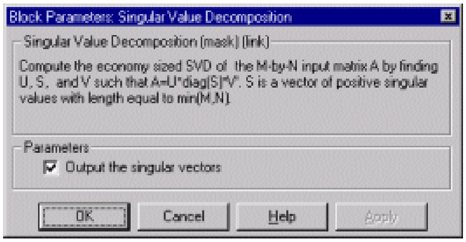
Length-N row inputs are treated as length-N columns.

```
[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



Block Parameters: Singular Value Decomposition

Singular Value Decomposition (mask) (link)

Compute the economy sized SVD of the M-by-N input matrix A by finding U, S, and V such that A=U*diag(S)*V'. S is a vector of positive singular values with length equal to min(M,N).

Parameters

☑ Output the singular vectors

Output the singular vectors

Enables the `U` and `V` output ports when selected.

References

Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

Supported Data Types

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

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

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 5-8 for related information.

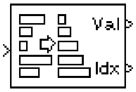
Purpose

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 jth column of idx is an index vector that permutes the jth column of u to the desired sorting order.

```
val(:,j) = u(idx(:,j),j)
```

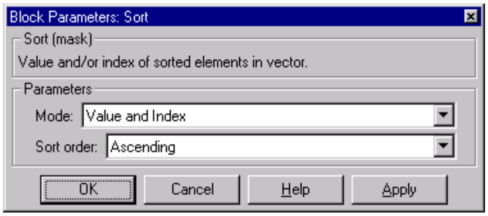
The index value outputs are always 32-bit unsigned integer values.

As in Value mode, length-M 1-D vector inputs and *sample-based* length-M row vector inputs are both treated as M-by-1 column vectors.

Value and Index Mode

When **Mode** is set to `Value and Index`, the block outputs both the sorted matrix, `val`, and the index matrix, `idx`.

Dialog Box



Block Parameters: Sort

Sort (mask)

Value and/or index of sorted elements in vector.

Parameters

Mode: Value and Index

Sort order: Ascending

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 training points, `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 your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-2.

See Also

Histogram	DSP Blockset
Median	DSP Blockset
`sort`	MATLAB

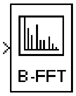
Purpose

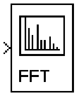
Compute and display the short-time FFT of each input signal

Library

DSP Sinks

Description





The Spectrum Scope block computes and displays the short-time 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** parameter.

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. We do not recommend 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 the Short-Time FFT block for more information.

In order to correctly scale the frequency axis (that is, 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 increment 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.

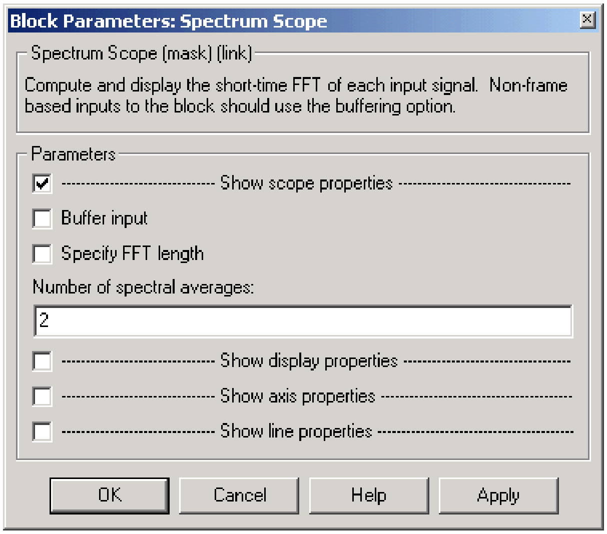
One example when these conditions do not hold, is such as when the input to the block is not the original signal, but a zero-padded or otherwise rate-altered version. In such cases, you should specify the appropriate value for the **Sample time of original time-series** parameter.

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.

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



Block Parameters: Spectrum Scope

Spectrum Scope (mask) (link)

Compute and display the short-time FFT of each input signal. Non-frame based inputs to the block should use the buffering option.

Parameters

☑ ------------------------------ Show scope properties ----------------------------------

□ Buffer input

□ Specify FFT length

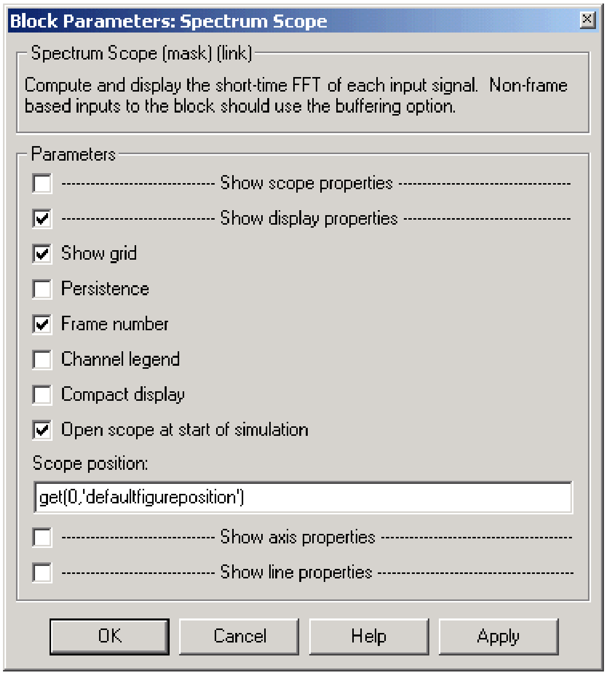
Number of spectral averages:

2

□ ------------------------------ Show display properties ---------------------------------

□ ------------------------------ Show axis properties ------------------------------------

□ ------------------------------ Show line properties -------------------------------------



Block Parameters: Spectrum Scope

Spectrum Scope (mask) (link)

Compute and display the short-time FFT of each input signal. Non-frame based inputs to the block should use the buffering option.

Parameters

□ ------------------------------ Show scope properties ----------------------------------

☑ ------------------------------ Show display properties ---------------------------------

☑ Show grid

□ Persistence

☑ Frame number

□ Channel legend

□ Compact display

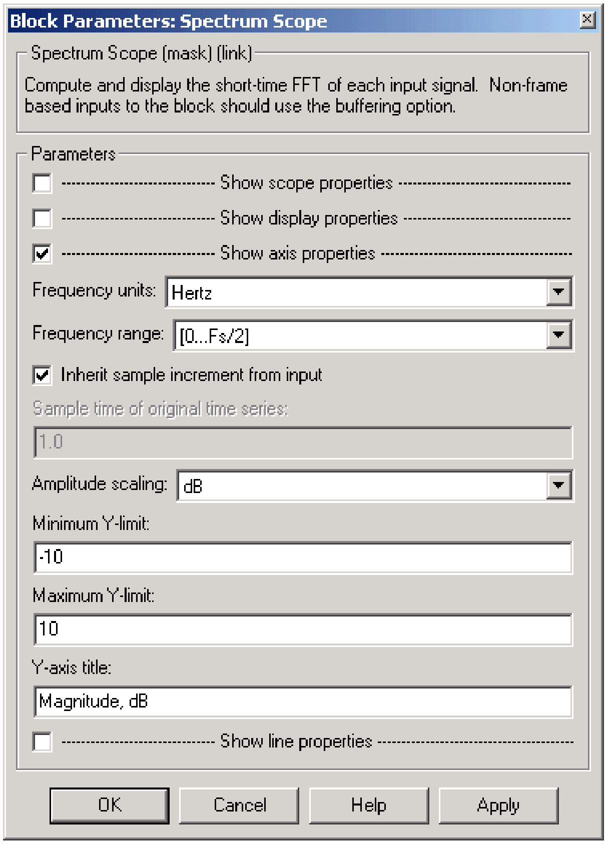
☑ Open scope at start of simulation

Scope position:

get(0,'defaultfigureposition')

□ ------------------------------ Show axis properties -------------------------------------

□ ------------------------------ Show line properties --------------------------------------



Block Parameters: Spectrum Scope

Spectrum Scope (mask) (link)

Compute and display the short-time FFT of each input signal. Non-frame based inputs to the block should use the buffering option.

Parameters

□ ------------------------------ Show scope properties ----------------------------------

□ ----------------------------- Show display properties --------------------------------

☑ ------------------------------ Show axis properties -------------------------------------

Frequency units: Hertz

Frequency range: [0...Fs/2]

☑ Inherit sample increment from input

Sample time of original time series:

1.0

Amplitude scaling: dB

Minimum Y-limit:

-10

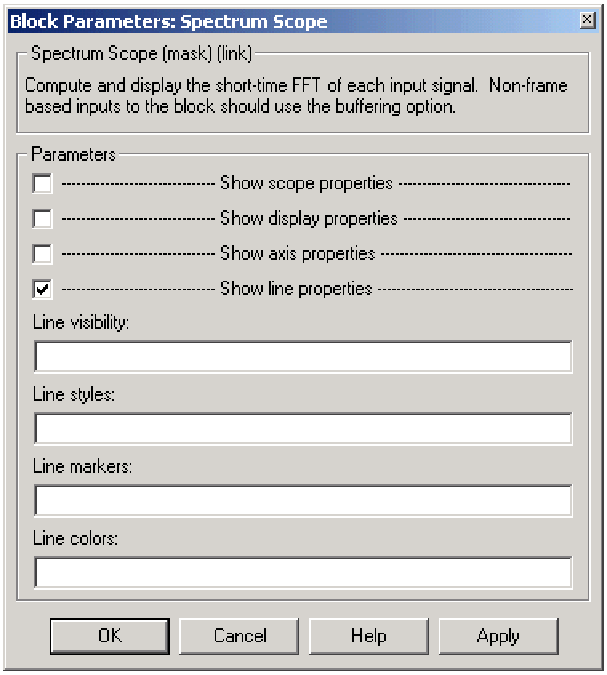
Maximum Y-limit:

10

Y-axis title:

Magnitude, dB

□ ----------------------------- Show line properties --------------------------------------



Block Parameters: Spectrum Scope

Spectrum Scope (mask) (link)

Compute and display the short-time FFT of each input signal. Non-frame based inputs to the block should use the buffering option.

Parameters

□ Show scope properties

□ Show display properties

□ Show axis properties

☑ Show line properties

Line visibility:

Line styles:

Line markers:

Line colors:

Show scope properties

Select to expose **Scope properties** panel.

Buffer input

Select to expose **Buffer input** panel.

Buffer size

The number of signal samples to include in each buffer.

Buffer overlap

The number of samples by which consecutive buffers overlap.

Specify FFT length

Select to expose **Specify FFT length** panel.

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.

Number of spectral averages

The number of spectra to average. Setting this parameter to 1 effectively disables averaging. See the Short-Time FFT block for more information.

Show display properties

Select to expose the **Display properties** panel. See the Vector Scope block for more information.

Show axis properties

Select to expose the **Axis properties** panel. See the Vector Scope block for more information.

Show line properties

Select to expose the **Line properties** panel. See the Vector Scope block for more information.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point
- Custom data types
- Boolean
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

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

See Also

FFT	DSP Blockset
Vector Scope	DSP Blockset

Also see “Viewing Signals” on page 2-87 for how to use this and other blocks to view signals.

Stack

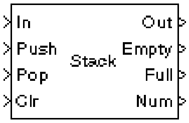
Purpose

Store inputs into a LIFO register.

Library

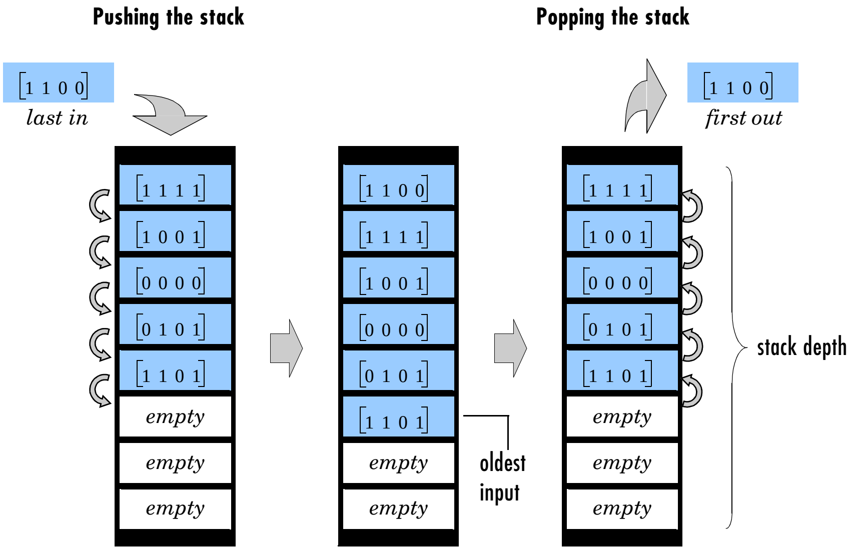
Signal Management / Buffers

Description



The Stack block stores a sequence of input samples in a last in, first out (LIFO) register. The register capacity is set by the **Stack depth** parameter, and inputs can be scalars, vectors, or matrices.

The block *pushes* the input at the `In` port onto the top of the stack when a trigger event is received at the `Push` port. When a trigger event is received at the `Pop` port, the block *pops* the top element off the stack and holds the `Out` port at that value. The last input to be pushed onto the stack is always the first to be popped off.



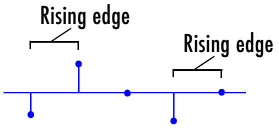
A trigger event at the optional `Clr` port (enabled by the **Clear input** check box) empties the stack contents. If **Clear output port on reset** is selected, then a trigger event at the `Clr` port empties the stack *and* sets the value at the `Out` port to zero. This setting also applies when a disabled subsystem containing the Stack block is reenabled; the `Out` port value is only reset to zero in this case if **Clear output port on reset** is selected.

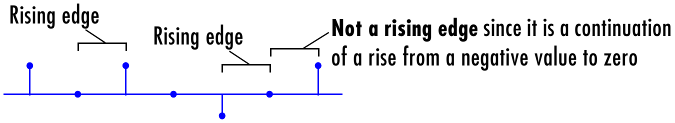
When two or more of the control input ports are triggered at the same time step, the operations are executed in the following order:

1. `Clr`
2. `Push`
3. `Pop`

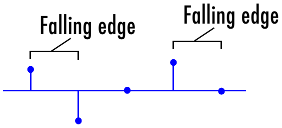
The rate of the trigger signal must be a positive integer multiple of the rate of the data signal input. The triggering event for the `Push`, `Pop`, and `Clr` ports is specified by the **Trigger type** pop-up menu, and can be one of the following:

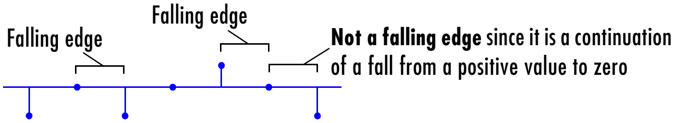
- `Rising edge` — Triggers execution of the block when the trigger input does one of the following:
  - Rises from a negative value to a positive value or zero
  - Rises from zero to a positive value, where the rise is not a continuation of a rise from a negative value to zero (see the following figure)




Rising edge
Rising edge
Not a rising edge since it is a continuation of a rise from a negative value to zero

- `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)




Falling edge
Falling edge
Not a falling edge since it is a continuation of a fall from a positive value to zero

- `Either edge` — Triggers execution of the block when the trigger input is a `Rising edge` or `Falling edge` (as described above).
- `Non-zero sample` — Triggers execution of the block at each sample time that the trigger input is not zero.

Note When running simulations in the Simulink `MultiTasking` mode, sample-based trigger signals have a one-sample latency, and frame-based trigger signals have one frame of latency. Thus, there is a one-sample or one-frame delay between the time the block detects a trigger event, and when it applies the trigger. For more information on latency and the Simulink tasking modes, see “Excess Algorithmic Delay (Tasking Latency)” on page 2-98 and the topic called 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

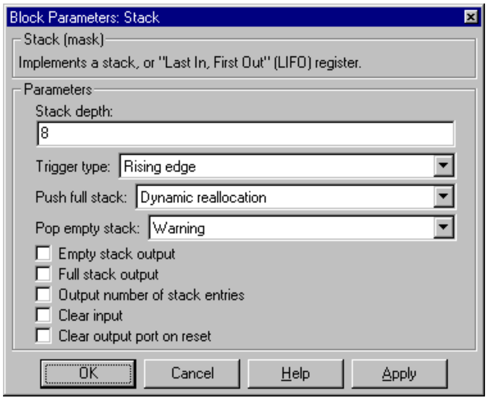
In	Push	Pop	Clr	Stack	Out	Empty	Full	Num
10	1	0	0	*top* 10 *bottom*	2	0	0	1
11	0	0	0	*top* 11 10 *bottom*	2	0	0	2
12	1	0	1	*top* 12 *bottom*	0	0	0	1

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



Block Parameters: Stack

Stack (mask)

Implements a stack, or "Last In, First Out" (LIFO) register.

Parameters

Stack depth: 8

Trigger type: Rising edge

Push full stack: Dynamic reallocation

Pop empty stack: Warning

☐ Empty stack output
☐ Full stack output
☐ Output number of stack entries
☐ Clear input
☐ Clear output port on reset

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-7. To learn how to disable Boolean output support, see “Steps to Disabling Boolean Support” on page A-8.
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

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

See Also

Buffer	DSP Blockset
Delay Line	DSP Blockset
Queue	DSP Blockset

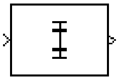
Purpose

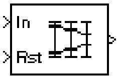
Find the standard deviation of an input or sequence of inputs

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

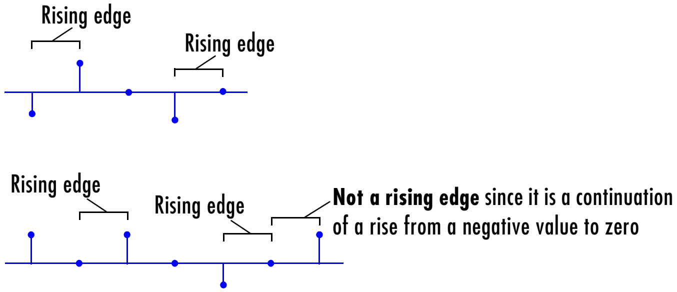
When the **Running standard deviation** check box is selected, the block tracks the standard deviation of each channel in a *time-sequence* of M-by-N inputs. For sample-based inputs, the output is a sample-based M-by-N matrix with each element y_{ij} containing the standard deviation of element u_{ij} 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.

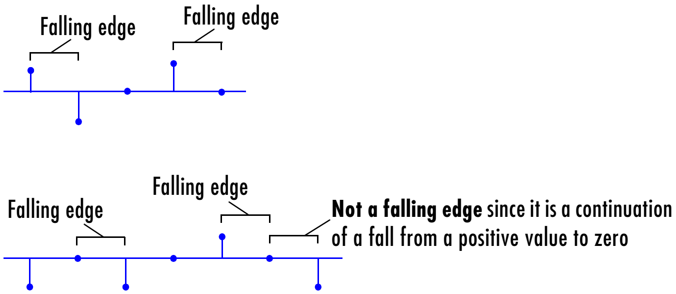
The reset event is specified by the **Reset port** parameter, and can be one of the following:

- `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)


Rising edge
Rising edge
Rising edge
Rising edge
Not a rising edge since it is a continuation
of a rise from a negative value to zero

- `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)

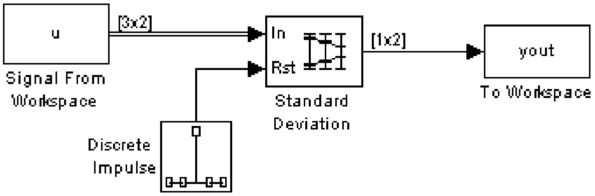

Falling edge
Falling edge
Falling edge
Falling edge
Falling edge
Not a falling edge since it is a continuation of a fall from a positive value to zero

- `Either edge` — Triggers a reset operation when the `Rst` input is a `Rising edge` or `Falling edge` (as described above).
- `Non-zero sample` — Triggers a reset operation at each sample time that the `Rst` input is not zero.

Note When running simulations in the Simulink `MultiTasking` mode, sample-based reset signals have a one-sample latency, and frame-based reset signals have one frame of latency. Thus, there is a one-sample or one-frame delay between the time the block detects a reset event, and when it applies the reset. For more information on latency and the Simulink tasking modes, see “Excess Algorithmic Delay (Tasking Latency)” on page 2-98 and the topic called 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`

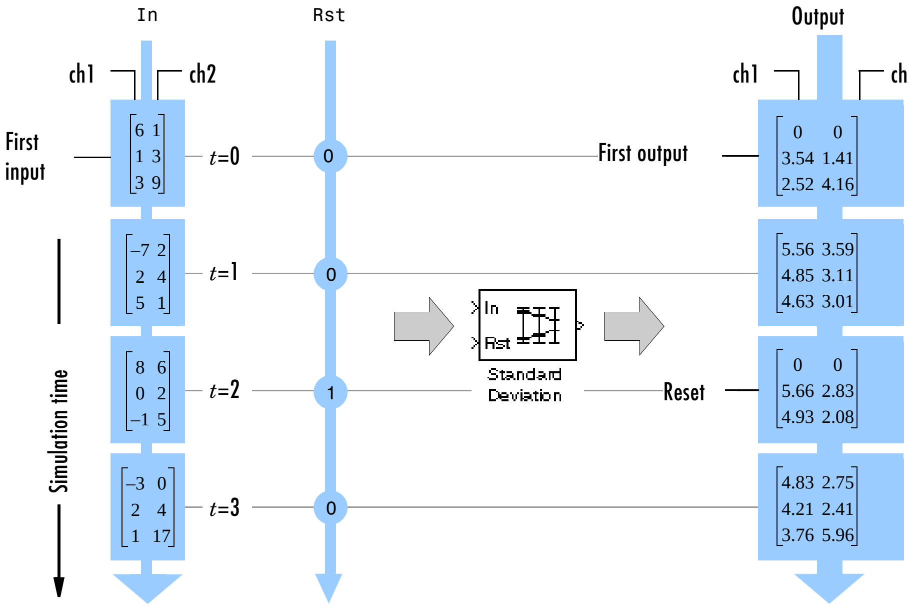
where

```
   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]'
```

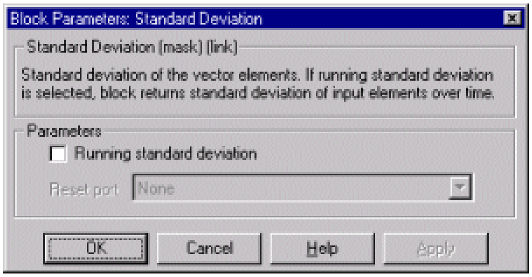
The Discrete Impulse block has the following settings:

- **Delay (samples)** = `2`
- **Sample time** = `1`
- **Samples per frame** = `1`

The block's operation is shown in the figure below.



Dialog Box



Block Parameters: Standard Deviation

Standard Deviation (mask) (link)

Standard deviation of the vector elements. If running standard deviation is selected, block returns standard deviation of input elements over time.

Parameters

□ Running standard deviation

Reset port: None

Running standard deviation

Enables running operation when selected.

Reset port

Determines the reset event that causes the block to reset the running standard deviation. The reset signal rate must be a positive integer multiple of the rate of the data signal input. This parameter is enabled only when you select **Running standard deviation**. For more information, see “Resetting the Running Standard Deviation” on page 7-712.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Boolean — The block accepts Boolean inputs to the `Rst` port.

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

See Also

Mean	DSP Blockset
RMS	DSP Blockset
Variance	DSP Blockset
`std`	MATLAB

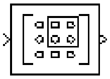
Purpose

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,:)`.

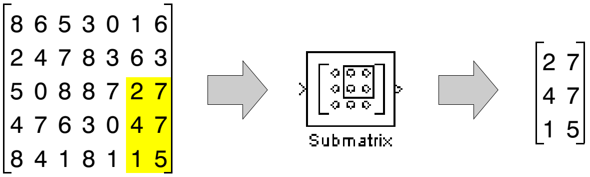
  For columns, this specifies that the middle column of `u` should be used as the last column of `y`. If all rows are to be included, this is equivalent to `y(:,end) = u(:,N/2)`.

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)`.

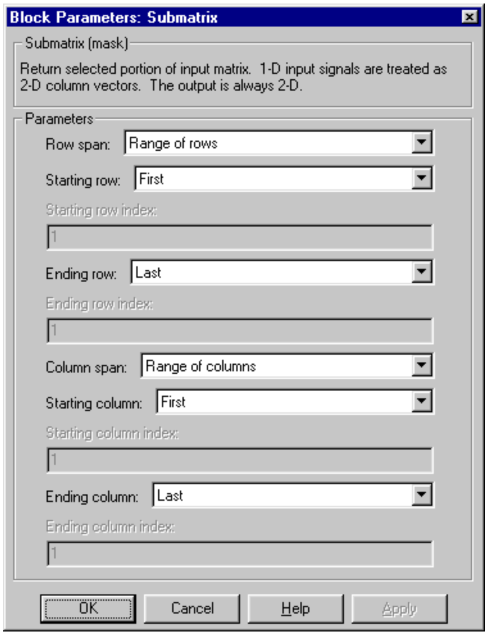


$$\begin{bmatrix} 8 & 6 & 5 & 3 & 0 & 1 & 6 \\ 2 & 4 & 7 & 8 & 3 & 6 & 3 \\ 5 & 0 & 8 & 8 & 7 & 2 & 7 \\ 4 & 7 & 6 & 3 & 0 & 4 & 7 \\ 8 & 4 & 1 & 8 & 1 & 1 & 5 \end{bmatrix} \qquad \begin{bmatrix} 2 & 7 \\ 4 & 7 \\ 1 & 5 \end{bmatrix}$$

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** = 7

Dialog Box



Block Parameters: Submatrix

Submatrix (mask)

Return selected portion of input matrix. 1-D input signals are treated as 2-D column vectors. The output is always 2-D.

Parameters

Row span: Range of rows

Starting row: First

Starting row index: 1

Ending row: Last

Ending row index: 1

Column span: Range of columns

Starting column: First

Starting column index: 1

Ending column: Last

Ending column index: 1

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 your data types to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-2.

See Also

Reshape	Simulink
Selector	Simulink
Variable Selector	DSP Blockset
`reshape`	MATLAB

See “Deconstructing Signals” on page 2-61 for related information.

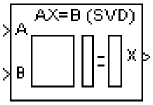
Purpose

Solve the equation AX=B using singular value decomposition

Library

Math Functions / Matrices and Linear Algebra / Linear System Solvers

Description

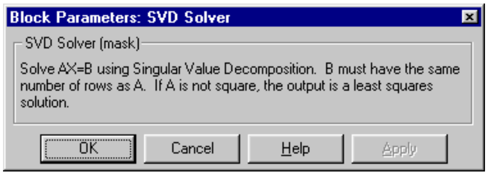


The SVD Solver block solves the linear system AX=B, which can be overdetermined, underdetermined, or exactly determined. The system is solved by applying singular value decomposition (SVD) factorization to the M-by-N matrix, A, at the A port. The input to the B port is the right side M-by-L matrix, B. A length-M 1-D vector input at either port is treated as an M-by-1 matrix.

The output at the x port is the N-by-L matrix, X. X is always sample based, and is chosen to minimize the sum of the squares of the elements of B-AX. When B is a vector, this solution minimizes the vector 2-norm of the residual (B-AX is the residual). When B is a matrix, this solution minimizes the matrix Frobenius norm of the residual. In this case, the columns of X are the solutions to the L corresponding systems $AX_k=B_k$, where B_k is the kth column of B, and X_k is the kth column of X.

X is known as the minimum-norm-residual solution to AX=B. The minimum-norm-residual solution is unique for overdetermined and exactly determined linear systems, but it is not unique for underdetermined linear systems. Thus when the SVD Solver block is applied to an underdetermined system, the output X is chosen such that the number of nonzero entries in X is minimized.

Dialog Box


Block Parameters: SVD Solver

SVD Solver (mask)

Solve AX=B using Singular Value Decomposition. B must have the same number of rows as A. If A is not square, the output is a least squares solution.

Supported Data Types

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

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

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 5-6 for related information.

Purpose

Display signals generated during a simulation

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 your data types to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-2.

Toeplitz

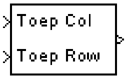
Purpose

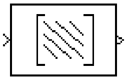
Generate a matrix with Toeplitz symmetry

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.

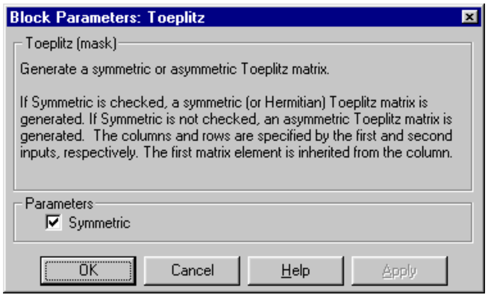
```
y = toeplitz(u)        % Equivalent MATLAB code
```

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



Block Parameters: Toeplitz

Toeplitz (mask)

Generate a symmetric or asymmetric Toeplitz matrix.

If Symmetric is checked, a symmetric (or Hermitian) Toeplitz matrix is generated. If Symmetric is not checked, an asymmetric Toeplitz matrix is generated. The columns and rows are specified by the first and second inputs, respectively. The first matrix element is inherited from the column.

Parameters

☑ Symmetric

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 your data types to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-2.

See Also

Constant Diagonal Matrix	DSP Blockset
`toeplitz`	MATLAB

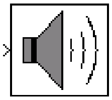
Purpose

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 pulse code modulation (PCM) format, and should typically be sampled at one of the standard Windows audio device rates: 8000, 11025, 22050, or 44100 Hz. Some hardware may support other rates in addition to these.

The **Use default audio device** check box allows the block to detect and use the system's default audio hardware. This option should be selected on systems that have a single sound device installed, or when the default sound device on a multiple-device system is the desired target. In cases when the default sound device is *not* the desired output device, clear **Use default audio device**, and set the desired audio device in the **Audio device** parameter, which lists the names of the installed audio device drivers.

The input to the block, u, can contain audio data from a mono or stereo signal. A mono signal is represented as either a sample-based scalar or frame-based length-M vector, while a stereo signal is represented as a sample-based length-2 vector or frame-based M-by-2 matrix. 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** check box.

```
sound(u,Fs,bits)       % Equivalent MATLAB code
```

Note that the block does not support `uint16` or `int8` data types. The 16-bit sample width requires more memory but in general yields better fidelity. The amplitude of the input must be in a valid range that depends on the input data type (see the following table). Amplitudes outside the valid range are clipped to the nearest allowable value.

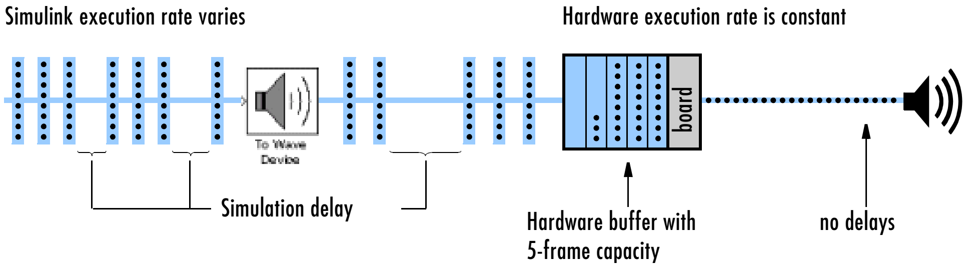
Input Data Type	Valid Input Amplitude Range
`double`	±1
`single`	±1
`int16`	-32768 to 32767 (-2^{15} to $2^{15} - 1$)
`uint8`	0 to 255

Buffering

Because the audio device generates real-time audio output, Simulink must maintain a continuous flow of data to the device throughout the simulation. Delays in passing data to the audio hardware can result in hardware errors or distortion of the output. This means that the To Wave Device block must in principle supply data to the audio hardware as quickly as the hardware reads the data. However, the To Wave Device block often *cannot* match the throughput rate of the audio hardware, especially when the simulation is running from within Simulink rather than as generated code. (Simulink execution speed routinely varies during the simulation as the host operating system services other processes.) The block must therefore rely on a buffering strategy to ensure that signal data is accessible to the hardware on demand.

At the start of the simulation, the To Wave Device block writes T_d seconds worth of signal data to the device (hardware) buffer, where 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 eight samples per frame. The buffer of the sound board has a five-frame capacity, not fully used at the instant shown. (If the signal sample rate was 8 kHz, for instance, this small buffer could hold approximately 0.005 second of data.)


Simulink execution rate varies
Hardware execution rate is constant
To Wave Device
board
Simulation delay
Hardware buffer with 5-frame capacity
no delays

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 (that is, data is not thrown away). More typically, the hardware throughput rate is higher than the simulation throughput rate, and the buffer tends to empty over the duration of the simulation.

Under normal operation, an empty buffer indicates that the simulation is finished, and the entire length of the audio signal has been processed. However, 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

  $$\frac{T_b F_s}{M_o}$$

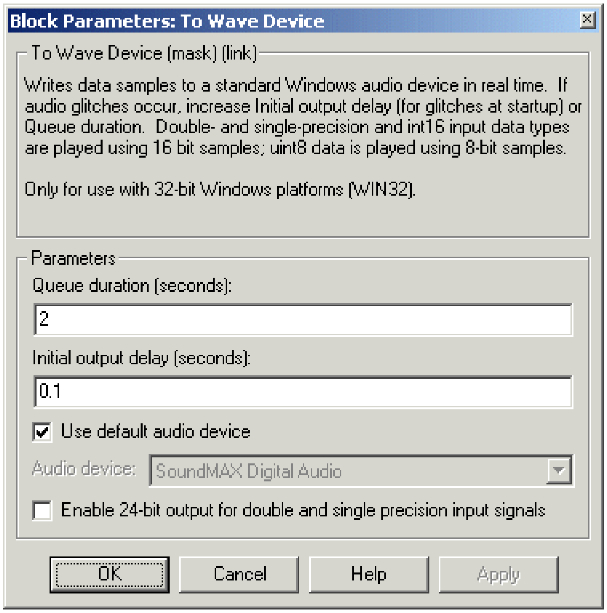
  where F_s is the sample rate of the signal and M_o is the number of samples per frame. The optimal buffer size for a given signal depends on the signal length, the frame size, and the speed of the simulation. The maximum number of frames that can be buffered is 1024.

- *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 2-92 for other ideas on improving simulation performance.

Dialog Box



To Wave Device (mask) (link)

Writes data samples to a standard Windows audio device in real time. If audio glitches occur, increase Initial output delay (for glitches at startup) or Queue duration. Double- and single-precision and int16 input data types are played using 16 bit samples; uint8 data is played using 8-bit samples.

Only for use with 32-bit Windows platforms (WIN32).

Parameters

Queue duration (seconds):

2

Initial output delay (seconds):

0.1

☑ Use default audio device

Audio device: SoundMAX Digital Audio

□ Enable 24-bit output for double and single precision input signals

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

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 2-86 for related information.

To Wave File

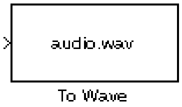
Purpose

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



The To Wave File block writes audio data to a Microsoft Wave (`.wav`) file in the uncompressed pulse code modulation (PCM) format. For compatibility reasons, the sample rate of the discrete-time input signal should typically be one of the standard Windows audio device rates (8000, 11025, 22050, or 44100 Hz), although the block supports arbitrary rates.

The input to the block, *u*, can contain audio data from a mono or stereo signal. A mono signal is represented as either a sample-based scalar or frame-based length-M vector, while a stereo signal is represented as a sample-based length-2 vector or frame-based M-by-2 matrix. The amplitude of the input should be in the range ±1. Values outside this range are clipped to the nearest allowable value.

```
wavwrite(u,Fs,bits,'filename')        % Equivalent MATLAB code
```

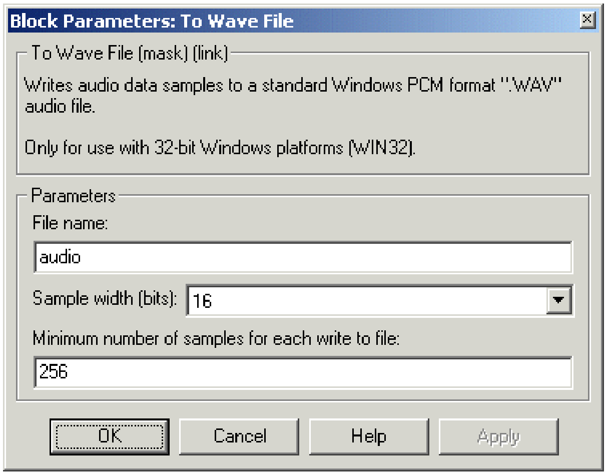
The **Sample Width (bits)** parameter specifies the number of bits used to represent the signal samples in the file. These settings are available:

- `8` — allocates 8 bits to each sample, allowing a resolution of 256 levels
- `16` — allocates 16 bits to each sample, allowing a resolution of 65536 levels
- `24` — allocates 24 bits to each sample, allowing a resolution of 16777216 levels
- `32` — allocates 32 bits to each sample, allowing a resolution of 232 levels ranging from -1 to 1

The higher sample width settings require more memory but yield better fidelity for double- and single-precision inputs.

The **File name** parameter can specify an absolute or relative path to the file. You do not need to specify the`.wav` extension. To reduce the required number of file accesses, the block writes L consecutive samples to the file during each access, where 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



Block Parameters: To Wave File

To Wave File (mask) (link)

Writes audio data samples to a standard Windows PCM format ".WAV" audio file.

Only for use with 32-bit Windows platforms (WIN32).

Parameters

File name: audio

Sample width (bits): 16

Minimum number of samples for each write to file: 256

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

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 2-86 for related information.

Transpose

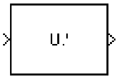
Purpose

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
```

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \end{bmatrix} \xrightarrow{u'} \begin{bmatrix} u_{11}^* & u_{21}^* \\ u_{12}^* & u_{22}^* \\ u_{13}^* & u_{23}^* \end{bmatrix}$$

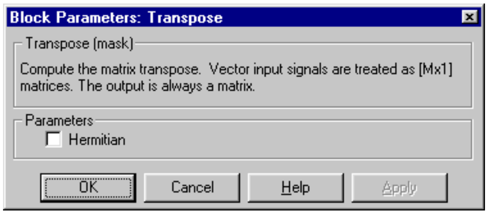
When the **Hermitian** check box is not selected, the block performs the nonconjugate transpose.

```
y = u.'         % Equivalent MATLAB code
```

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \end{bmatrix} \xrightarrow{u.'} \begin{bmatrix} u_{11} & u_{21} \\ u_{12} & u_{22} \\ u_{13} & u_{23} \end{bmatrix}$$

A length-M 1-D vector input is treated as an M-by-1 matrix. The output is always sample based.

Dialog Box


Block Parameters: Transpose

Transpose (mask)

Compute the matrix transpose. Vector input signals are treated as [Mx1] matrices. The output is always a matrix.

Parameters

□ Hermitian

Hermitian

When selected, specifies the complex conjugate transpose. 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 your data types to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-2.

See Also

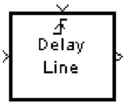
Permute Matrix	DSP Blockset
Reshape	Simulink
Submatrix	DSP Blockset

Triggered Delay Line

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.

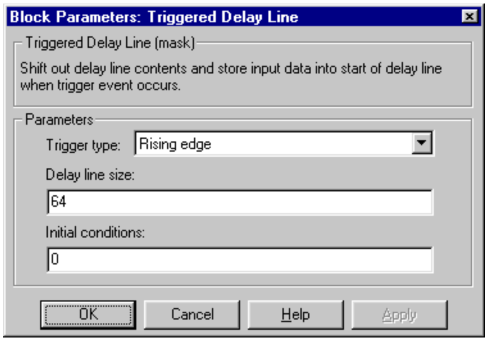
Frame-Based Operation

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 (that is, the number of consecutive samples from the input frame to rebuffer into the output frame). M_o can be greater or less than the input frame size, M_i. Each of the N input channels is rebuffered independently.

Initial Conditions

The Triggered Delay Line block's buffer is initialized to the value specified by the **Initial condition** parameter. The block always outputs this buffer at the first simulation step (t=0). 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



Block Parameters: Triggered Delay Line

Triggered Delay Line (mask)

Shift out delay line contents and store input data into start of delay line when trigger event occurs.

Parameters

Trigger type: Rising edge

Delay line size: 64

Initial conditions: 0

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 your data types to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-2.

See Also

Buffer	DSP Blockset
Delay Line	DSP Blockset
Unbuffer	DSP Blockset

Triggered Signal From Workspace

Purpose

Import signal samples from the MATLAB workspace when triggered

Library

Signal Operations

Description

The Triggered Signal From Workspace block imports signal samples from the MATLAB workspace into the Simulink model when triggered by the control signal at the input port (↑). The **Signal** parameter specifies the name of a MATLAB workspace variable containing the signal to import, or any valid MATLAB expression defining a matrix or 3-D array.

When the **Signal** parameter specifies an M-by-N matrix (M≠1), each of the N columns is treated as a distinct channel. 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.

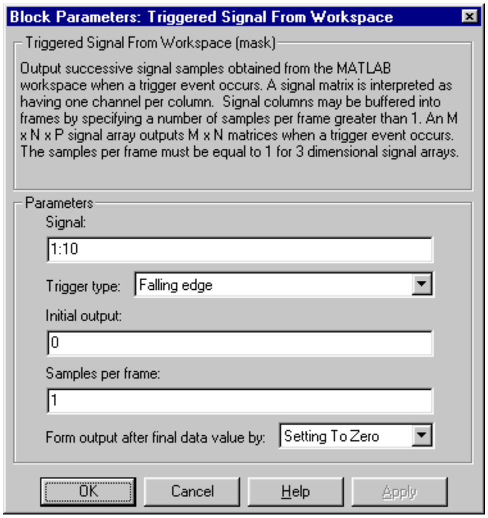
Initial and Final Conditions

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 (that is, no linear interpolation takes place). For single-channel signals, the **Initial output** parameter value can be a vector of length M_o or a scalar to repeat across the M_o elements of the initial output frames. For matrix outputs (M_o-by-N or M-by-N), the **Initial output** parameter value can be a vector of length N to repeat across all rows of the initial outputs, or a scalar to repeat across all elements of the initial matrix outputs.

When the block has output all of the available signal samples, it can start again at the beginning of the signal, or simply repeat the final value or generate zeros until the end of the simulation. (The block does not extrapolate the imported signal beyond the last sample.) The **Form output after final data value by** parameter controls this behavior:

- 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 (for example, the *i*th frame of one cycle contains the same samples as the *i*th frame of any other cycle).

Dialog Box



Triggered Signal From Workspace (mask)

Output successive signal samples obtained from the MATLAB workspace when a trigger event occurs. 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 when a trigger event occurs. The samples per frame must be equal to 1 for 3 dimensional signal arrays.

Parameters

Signal: 1:10

Trigger type: Falling edge

Initial output: 0

Samples per frame: 1

Form output after final data value by: Setting To Zero

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 your data types to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-2.

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 2-2
- “Multichannel Signals” on page 2-10
- “Benefits of Frame-Based Processing” on page 2-13
- “Creating Signals Using the Signal From Workspace Block” on page 2-37
- “Importing Signals” on page 2-69

Triggered To Workspace

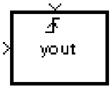
Purpose

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. The block overwrites an existing variable with the same name.

For an M-by-N frame-based input, the block creates an N-column workspace matrix in which each group of M rows represents a single input frame from each of N channels (the most recent frame occupying the last M rows). The maximum size of this workspace variable is limited to P-by-N, where P is the **Maximum number of rows** parameter. (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.

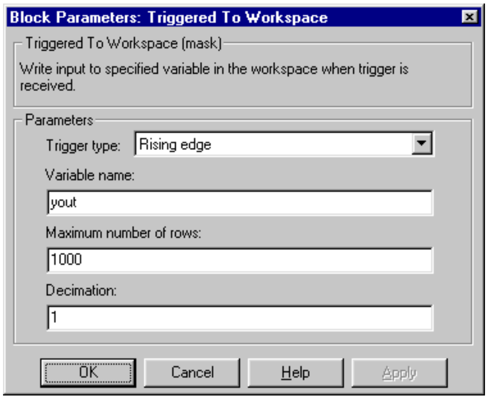
The block acquires and buffers a single frame from input 1 whenever it is triggered by the control signal at input 2 (↑). At all other times, the block ignores input 1. The triggering event at input 2 is specified by the **Trigger type** pop-up menu, and can be one of the following:

- `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, select 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



Block Parameters: Triggered To Workspace

Triggered To Workspace (mask)

Write input to specified variable in the workspace when trigger is received.

Parameters

Trigger type: Rising edge

Variable name: yout

Maximum number of rows: 1000

Decimation: 1

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

See Also

Signal From Workspace	DSP Blockset
To Workspace	Simulink

See "Exporting Signals" on page 2-79 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

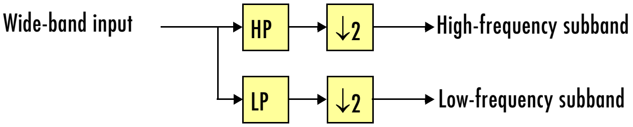
Description

Hi band
Lo band

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 7-756.

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 7-759.

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 7-755.

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 7-755.
- `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 2-98 and the topic called The Simulation Parameters Dialog Box in the Simulink documentation.

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`

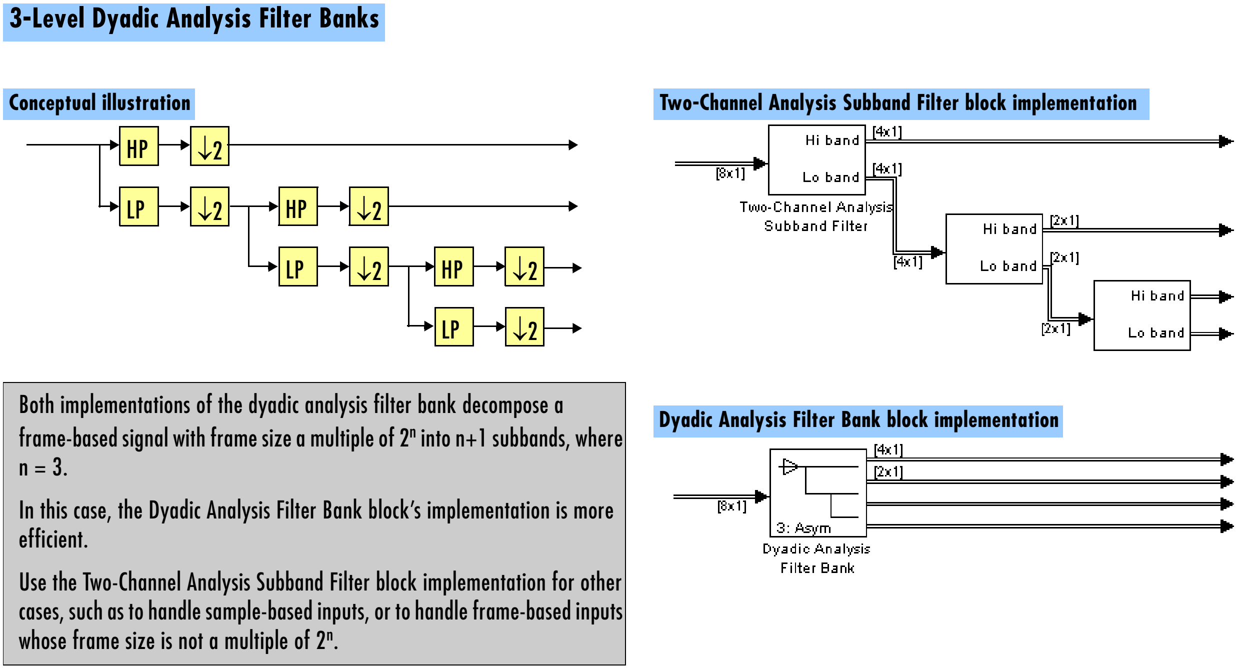
Creating Multilevel Dyadic Analysis Filter Banks

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 the Dyadic Analysis Filter Bank block 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 the previous table called Amount of Block Latency for All Possible Block Settings.)

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.



3-Level Dyadic Analysis Filter Banks

Conceptual illustration

Two-Channel Analysis Subband Filter block implementation

Both implementations of the dyadic analysis filter bank decompose a frame-based signal with frame size a multiple of 2^n into n+1 subbands, where n = 3.

In this case, the Dyadic Analysis Filter Bank block's implementation is more efficient.

Use the Two-Channel Analysis Subband Filter block implementation for other cases, such as to handle sample-based inputs, or to handle frame-based inputs whose frame size is not a multiple of 2^n.

Dyadic Analysis Filter Bank block implementation

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:

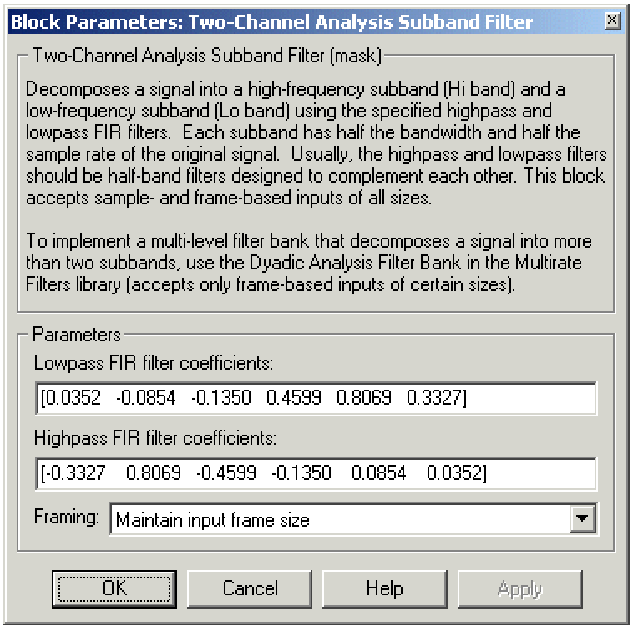
- Multi-level PR filter bank
- Denoising
- Wavelet transmultiplexer (WTM)

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



Block Parameters: Two-Channel Analysis Subband Filter

Two-Channel Analysis Subband Filter (mask)

Decomposes a signal into a high-frequency subband (Hi band) and a low-frequency subband (Lo band) using the specified highpass and lowpass FIR filters. Each subband has half the bandwidth and half the sample rate of the original signal. Usually, the highpass and lowpass filters should be half-band filters designed to complement each other. This block accepts sample- and frame-based inputs of all sizes.

To implement a multi-level filter bank that decomposes a signal into more than two subbands, use the Dyadic Analysis Filter Bank in the Multirate Filters library (accepts only frame-based inputs of certain sizes).

Parameters

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]

Framing: Maintain input frame size

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 specify 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 7-753.

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 specify 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 7-753.

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 7-754. Some settings of this parameter causes the block to have nonzero latency, as described in "Latency" on page 7-755.

References

Fliege, N. J. *Multirate Digital Signal Processing: Multirate Systems, Filter Banks, Wavelets*. West Sussex, England: John Wiley & Sons, 1994.

Strang, G. and T. Nguyen. *Wavelets and Filter Banks*. Wellesley, MA: Wellesley-Cambridge Press, 1996.

Vaidyanathan, P. P. *Multirate Systems and Filter Banks*. Englewood Cliffs, NJ: Prentice Hall, 1993.

Supported Data Types

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

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

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 3-61.

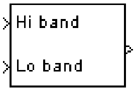
Purpose

Reconstruct a signal from a high-frequency subband and a low-frequency subband

Library

Filtering / Multirate Filters

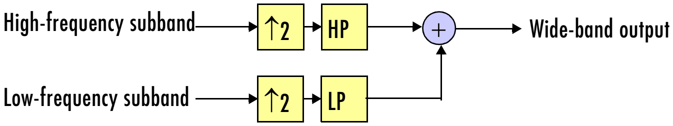
Description



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 7-765.

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)]`.

$$H(z) = B(z) = b_1 + b_2 z^{-1} + \dots + b_m z^{-(m-1)}$$

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 7-768.

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 M-by-N matrix as MxN independent subbands, where MxN is the product of the matrix dimensions. Each matrix element is the high- or low-frequency subband of the corresponding channel in the output matrix. The input to the topmost input port should contain the high-frequency subbands.

Sample-Based Outputs. Given valid sample-based inputs, the block outputs one sample-based matrix with the same dimensions as the inputs. The output sample rate is twice that of the input. Each element of the output is a single channel, reconstructed from the corresponding elements in each input matrix. Depending on the Simulink simulation parameters, some sample-based outputs may have one sample of latency, as described in "Latency" on page 7-764.

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 7-755.
- `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 2-98 and the topic called The Simulation Parameters Dialog Box in the Simulink documentation.

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`

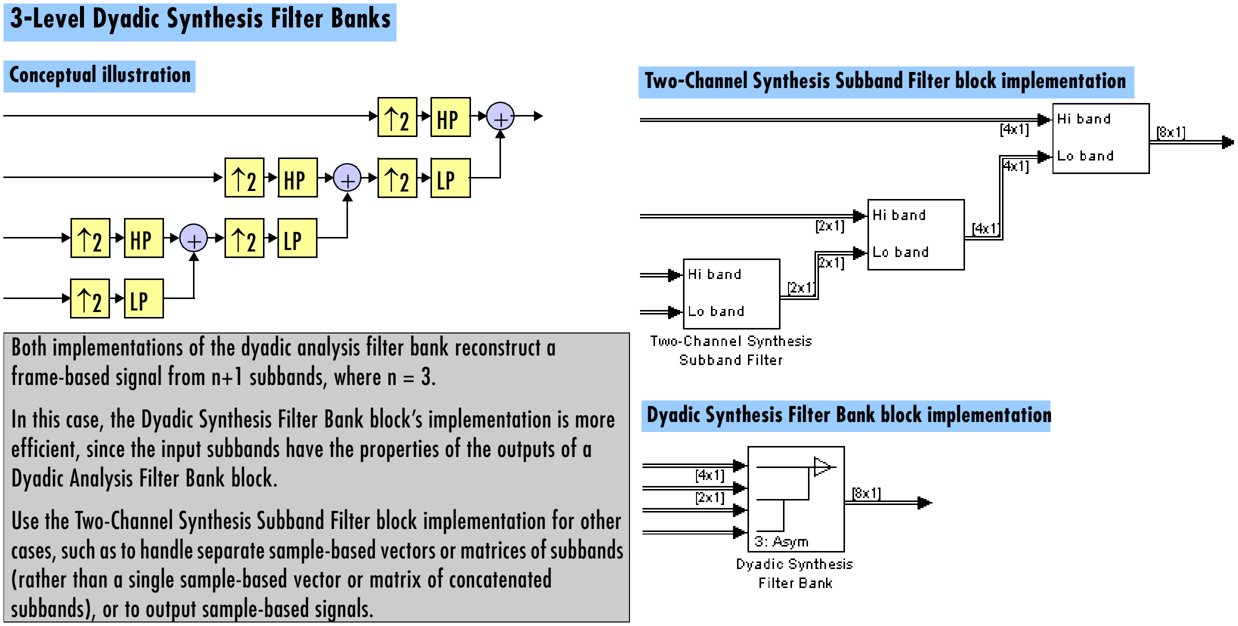
Creating Multilevel Dyadic Synthesis Filter Banks

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 the Dyadic Synthesis Filter Bank block 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 the previous table called Amount of Block Latency for All Possible Block Settings.)

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 block to reconstruct a frame-based signal (with frame size a multiple of 2^n) from 2^n or n+1 subbands whose properties match those of the Dyadic Analysis Filter Bank block's outputs. These properties are described in the Dyadic Analysis Filter Bank reference page.


3-Level Dyadic Synthesis Filter Banks
Conceptual illustration
↑2
HP
LP
+
Two-Channel Synthesis Subband Filter block implementation
Hi band
Lo band
[4x1]
[8x1]
[2x1]
Two-Channel Synthesis Subband Filter
Dyadic Synthesis Filter Bank block implementation
3: Asym
Dyadic Synthesis Filter Bank

Both implementations of the dyadic analysis filter bank reconstruct a frame-based signal from n+1 subbands, where n = 3.

In this case, the Dyadic Synthesis Filter Bank block's implementation is more efficient, since the input subbands have the properties of the outputs of a Dyadic Analysis Filter Bank block.

Use the Two-Channel Synthesis Subband Filter block implementation for other cases, such as to handle separate sample-based vectors or matrices of subbands (rather than a single sample-based vector or matrix of concatenated subbands), or to output sample-based signals.

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:

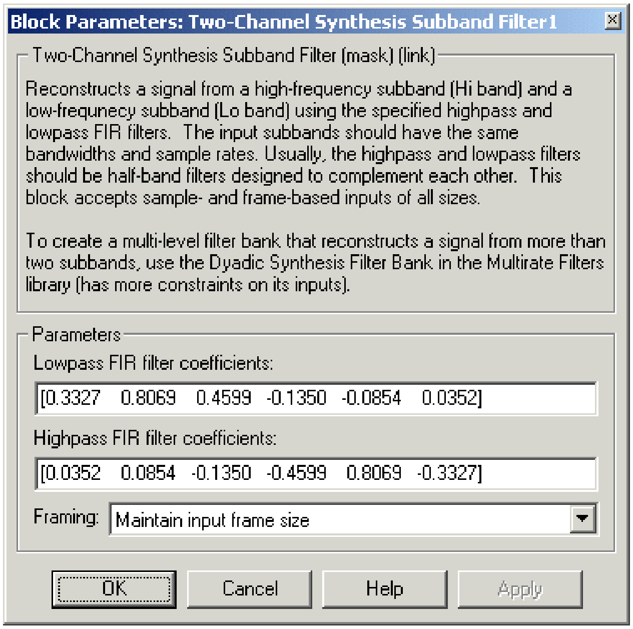
- Multi-level PR filter bank
- Denoising
- Wavelet transmultiplexer (WTM)

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



Block Parameters: Two-Channel Synthesis Subband Filter1

Two-Channel Synthesis Subband Filter (mask) (link)

Reconstructs a signal from a high-frequency subband (Hi band) and a low-frequnecy subband (Lo band) using the specified highpass and lowpass FIR filters. The input subbands should have the same bandwidths and sample rates. Usually, the highpass and lowpass filters should be half-band filters designed to complement each other. This block accepts sample- and frame-based inputs of all sizes.

To create a multi-level filter bank that reconstructs a signal from more than two subbands, use the Dyadic Synthesis Filter Bank in the Multirate Filters library (has more constraints on its inputs).

Parameters

Lowpass FIR filter coefficients: [0.3327 0.8069 0.4599 -0.1350 -0.0854 0.0352]

Highpass FIR filter coefficients: [0.0352 0.0854 -0.1350 -0.4599 0.8069 -0.3327]

Framing: Maintain input frame size

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 7-762.

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 7-762.

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 7-754. Some settings of this parameter causes the block to have nonzero latency, as described in "Latency" on page 7-755.

References

Fliege, N. J. *Multirate Digital Signal Processing: Multirate Systems, Filter Banks, Wavelets*. West Sussex, England: John Wiley & Sons, 1994.

Strang, G. and T. Nguyen. *Wavelets and Filter Banks*. Wellesley, MA: Wellesley-Cambridge Press, 1996.

Vaidyanathan, P. P. *Multirate Systems and Filter Banks*. Englewood Cliffs, NJ: Prentice Hall, 1993.

Supported Data Types

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

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

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 3-61.

Unbuffer

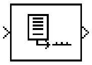
Purpose

Unbuffer a frame input to a sequence of scalar outputs

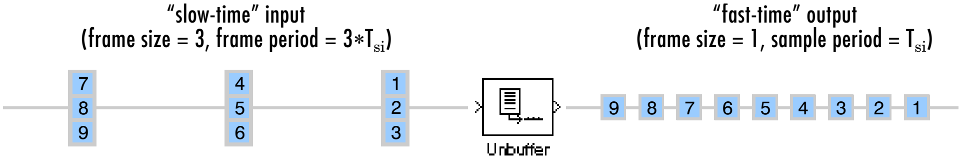
Library

Signal Management / Buffers

Description

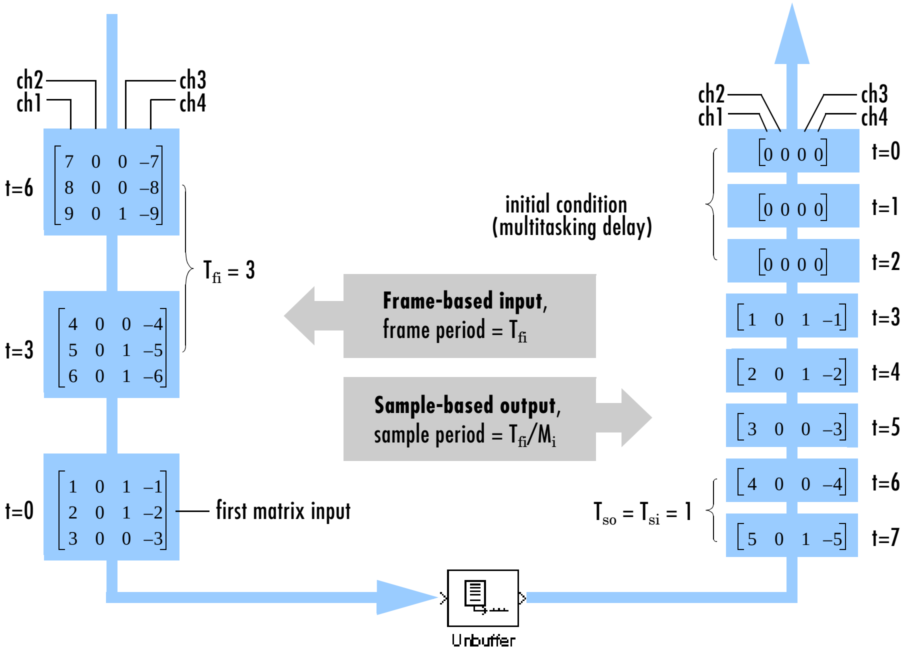


The Unbuffer block unbuffers an M_i-by-N frame-based input into a 1-by-N sample-based output. That is, inputs are unbuffered *row-wise* so that each matrix row becomes an independent time-sample in the output. The rate at which the block receives inputs is generally less than the rate at which the block produces outputs.


"slow-time" input
(frame size = 3, frame period = $3*T_{si}$)
"fast-time" output
(frame size = 1, sample period = T_{si})
7
8
9
4
5
6
1
2
3
Unbuffer
9
8
7
6
5
4
3
2
1

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.

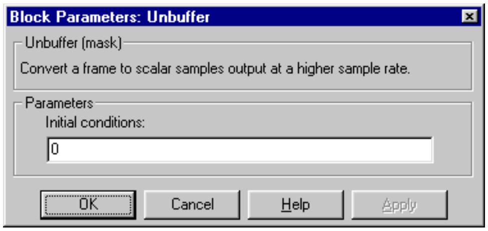
Nonzero Latency. For *multitasking* operation, the Unbuffer block's buffer is initialized with the value specified by the **Initial condition** parameter, and the block begins unbuffering this frame at the start of the simulation. Inputs to the block are therefore delayed by one buffer length, or M_i 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 2-98 and the topic called The Simulation Parameters Dialog Box in the Simulink documentation for more information about block rates and the Simulink tasking modes.

Dialog Box



Block Parameters: Unbuffer

Unbuffer (mask)

Convert a frame to scalar samples output at a higher sample rate.

Parameters

Initial conditions: 0

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 your data types to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-2.

See Also

Buffer	DSP Blockset

See “Unbuffering a Frame-Based Signal into a Sample-Based Signal” on page 2-67 for related information.

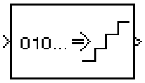
Purpose

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:

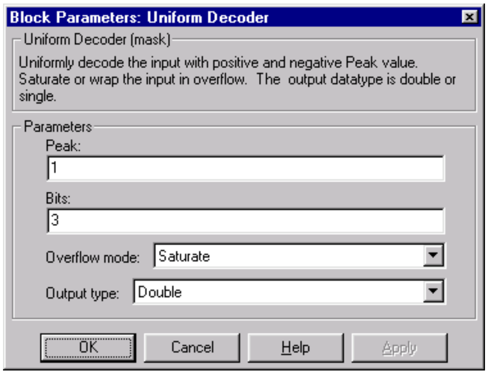
- **Peak** = `2`
- **Bits** = `3`

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 block reference page for more about these settings.)

The real and complex components of each input are independently mapped to one of 2^3 distinct levels in the range `[-2.0,1.5]`.

0	is mapped to	-2.0
1	is mapped to	-1.5
2	is mapped to	-1.0
3	is mapped to	-0.5
4	is mapped to	0.0
5	is mapped to	0.5
6	is mapped to	1.0
7	is mapped to	1.5

Dialog Box



Block Parameters: Uniform Decoder

Uniform Decoder (mask)

Uniformly decode the input with positive and negative Peak value. Saturate or wrap the input in overflow. The output datatype is double or single.

Parameters

Peak: 1

Bits: 3

Overflow mode: Saturate

Output type: Double

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

See Also

Data Type Conversion	Simulink
Quantizer	Simulink
Scalar Quantizer	DSP Blockset
Uniform Encoder	DSP Blockset
`udecode`	Signal Processing Toolbox
`uencode`	Signal Processing Toolbox

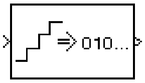
Purpose

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.

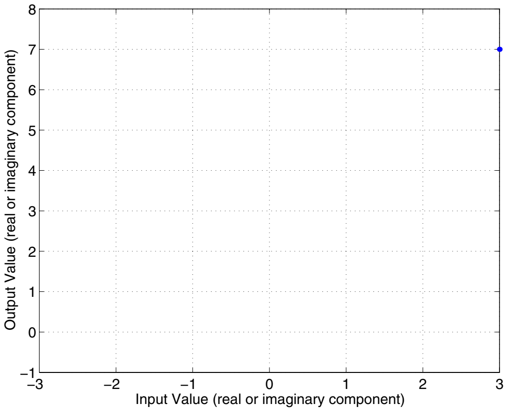
Bits	Unsigned Integer	Signed Integer
2 to 8	`uint8`	`int8`
9 to 16	`uint16`	`int16`
17 to 32	`uint32`	`int32`

The Uniform Encoder block operations adhere to the definition for uniform encoding specified in ITU-T Recommendation G.701.

Examples

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]`.

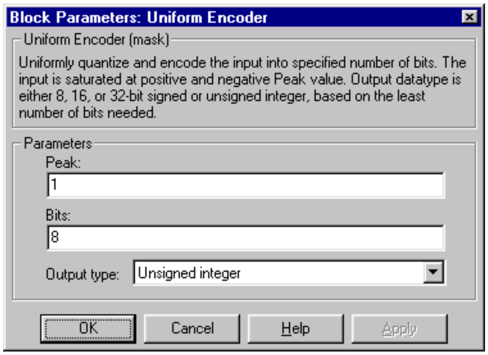
-2.0	is mapped to	0
-1.5	is mapped to	1
-1.0	is mapped to	2
-0.5	is mapped to	3
0.0	is mapped to	4
0.5	is mapped to	5
1.0	is mapped to	6
1.5	is mapped to	7

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 imaginary)

The output data type is automatically set to `uint8`, the most efficient format for this input range.

Dialog Box


Block Parameters: Uniform Encoder

Uniform Encoder (mask)

Uniformly quantize and encode the input into specified number of bits. The input is saturated at positive and negative Peak value. Output datatype is either 8, 16, or 32-bit signed or unsigned integer, based on the least number of bits needed.

Parameters

Peak: 1

Bits: 8

Output type: Unsigned integer

Peak

The largest input amplitude to be encoded, V. Real or imaginary input values greater than $(1\text{-}2^{1\text{-}B})$V or less than -V saturate (independently for complex inputs) at those limits.

Bits

The number of levels at which to quantize the floating-point input. (Also the number of bits needed to represent the integer output.)

Output type

The data type of the block's output, `Unsigned integer` or `Signed integer`. Unsigned outputs are `uint8`, `uint16`, or `uint32`, while signed outputs are `int8`, `int16`, or `int32`.

References

General Aspects of Digital Transmission Systems: Vocabulary of Digital Transmission and Multiplexing, and Pulse Code Modulation (PCM) Terms, International Telecommunication Union, ITU-T Recommendation G.701, March, 1993

Supported Data Types

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

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

See Also

Data Type Conversion	Simulink
Quantizer	Simulink
Scalar Quantizer	DSP Blockset
Uniform Decoder	DSP Blockset
`udecode`	Signal Processing Toolbox
`uencode`	Signal Processing Toolbox

Unwrap

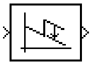
Purpose

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 7-782
- “The Two Unwrap Modes” on page 7-783
- “Unwrap Method” on page 7-786
- “Definition of Phase Unwrap” on page 7-786

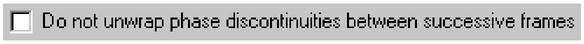
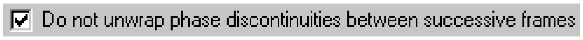
Acceptable Inputs and Corresponding Output Characteristics

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 7-786.) The number of times that k is reset to 0 depends on the unwrap mode.</td></tr>
<tr><td><b>Default Unwrap Mode:
Initialize <i>k</i> to 0 for Only the First Input Frame</b></td><td><b>Nondefault Unwrap Mode:
Set <i>k</i> to 0 for Each Successive Input Matrix or Input Vector</b></td></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>

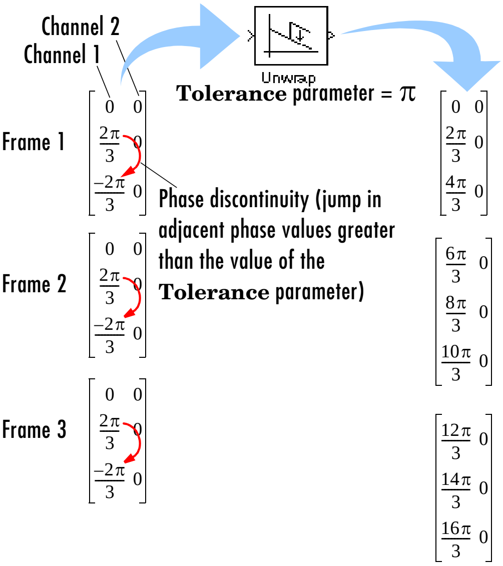
The following diagrams illustrate how the two unwrap modes operate on various inputs.

Default Unwrap Mode Operation:

□ Do not unwrap phase discontinuities between successive frames

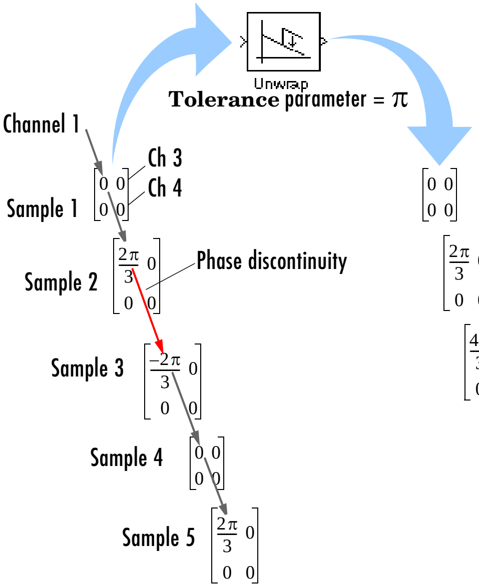
Frame-Based Inputs

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.


Channel 2
Channel 1
Unwrap
Tolerance parameter = π
Frame 1
Frame 2
Frame 3
Phase discontinuity (jump in adjacent phase values greater than the value of the Tolerance parameter)

Sample-Based Inputs

The block treats each element of the input matrix as an independent channel. (The first sample in Channel 1 is in the upper left corner of the Sample 1 matrix. The second sample of Channel 1 is in the corresponding corner of the Sample 2 matrix, and so on.)

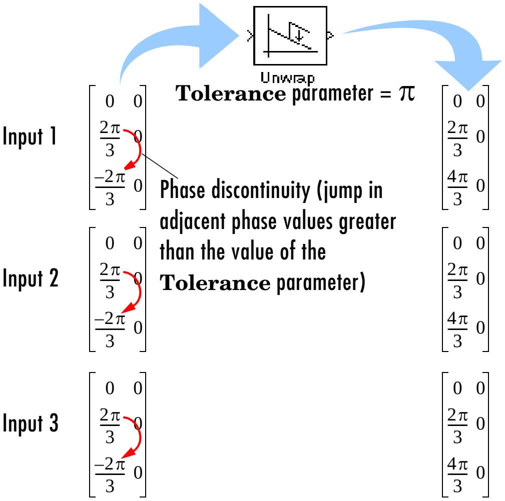


Nondefault Unwrap Mode Operation:

☑ Do not unwrap phase discontinuities between successive frames

Frame-Based Inputs and Sample-Based (Nonrow) Inputs

The block unwraps each column, treating each input matrix as completely unrelated to the other input matrices.

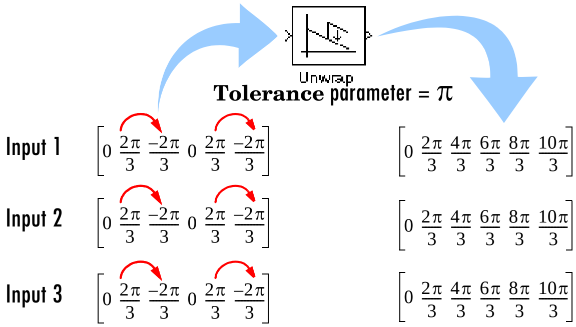

Unwrap
Tolerance parameter = π
Input 1
Input 2
Input 3
Phase discontinuity (jump in adjacent phase values greater than the value of the **Tolerance** parameter)

Input 1, Input 2, Input 3:

$$\begin{bmatrix} 0 & 0 \\ \frac{2\pi}{3} & 0 \\ \frac{-2\pi}{3} & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 0 \\ \frac{2\pi}{3} & 0 \\ \frac{4\pi}{3} & 0 \end{bmatrix}$$

Sample-Based Row Vector Inputs

The block unwraps each row, treating each input row vector as completely independent of the other input row vectors.



Input 1, Input 2, Input 3:

$$\begin{bmatrix} 0 & \frac{2\pi}{3} & \frac{-2\pi}{3} & 0 & \frac{2\pi}{3} & \frac{-2\pi}{3} \end{bmatrix} \quad \begin{bmatrix} 0 & \frac{2\pi}{3} & \frac{4\pi}{3} & \frac{6\pi}{3} & \frac{8\pi}{3} & \frac{10\pi}{3} \end{bmatrix}$$

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.

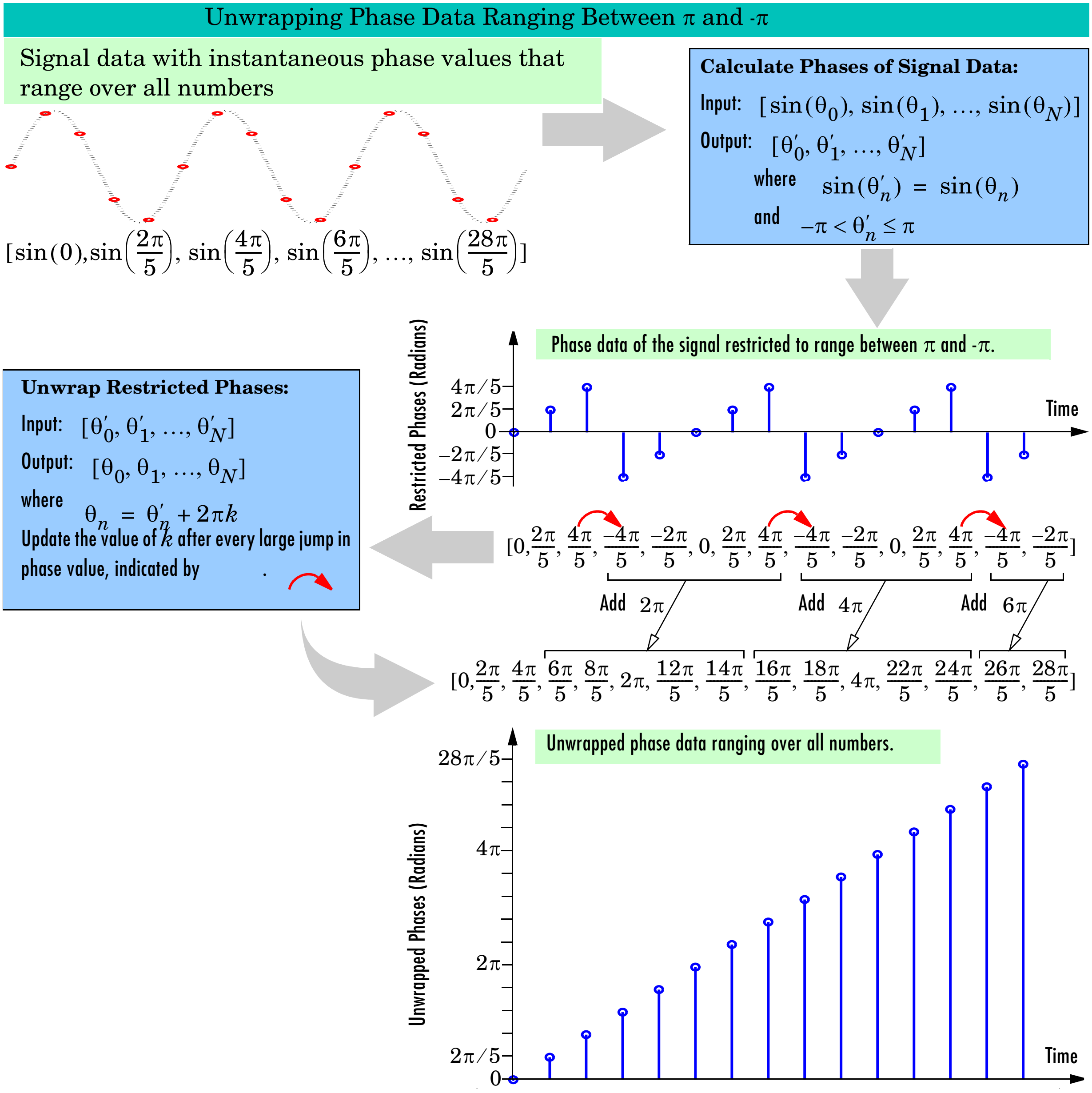
Steps to the Unwrap Method:

1. Set k to 0 (See “The Two Unwrap Modes” on page 7-783 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 7-786.



Unwrapping Phase Data Ranging Between π and -π

Signal data with instantaneous phase values that range over all numbers

$[\sin(0), \sin\left(\frac{2\pi}{5}\right), \sin\left(\frac{4\pi}{5}\right), \sin\left(\frac{6\pi}{5}\right), \ldots, \sin\left(\frac{28\pi}{5}\right)]$

Calculate Phases of Signal Data:

Input: $[\sin(\theta_0), \sin(\theta_1), \ldots, \sin(\theta_N)]$

Output: $[\theta'_0, \theta'_1, \ldots, \theta'_N]$

where $\sin(\theta'_n) = \sin(\theta_n)$

and $-\pi < \theta'_n \le \pi$

Phase data of the signal restricted to range between π and -π.

Unwrap Restricted Phases:

Input: $[\theta'_0, \theta'_1, \ldots, \theta'_N]$

Output: $[\theta_0, \theta_1, \ldots, \theta_N]$

where $\theta_n = \theta'_n + 2\pi k$

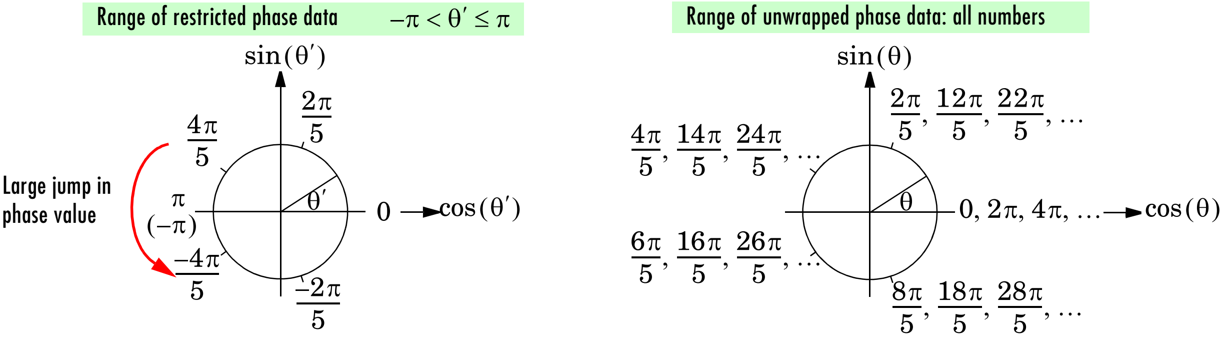
Update the value of k after every large jump in phase value, indicated by .

$[0, \frac{2\pi}{5}, \frac{4\pi}{5}, \frac{-4\pi}{5}, \frac{-2\pi}{5}, 0, \frac{2\pi}{5}, \frac{4\pi}{5}, \frac{-4\pi}{5}, \frac{-2\pi}{5}, 0, \frac{2\pi}{5}, \frac{4\pi}{5}, \frac{-4\pi}{5}, \frac{-2\pi}{5}]$

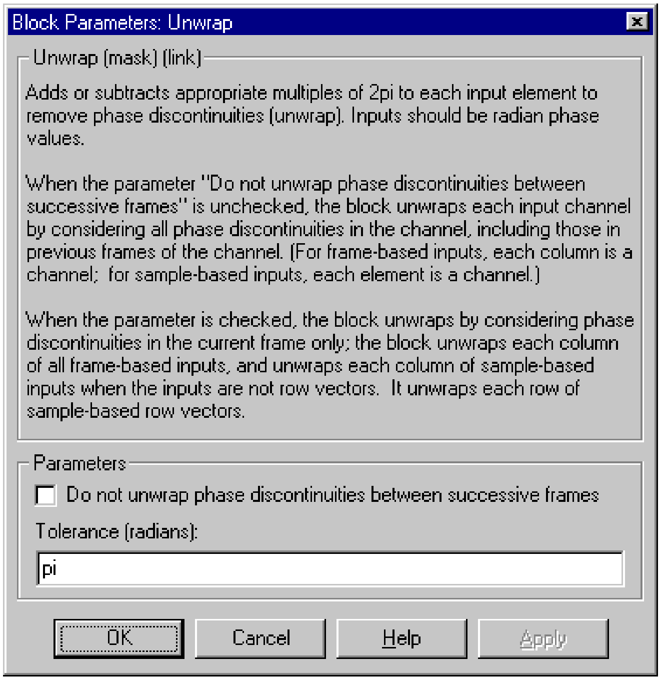
Add 2π Add 4π Add 6π

$[0, \frac{2\pi}{5}, \frac{4\pi}{5}, \frac{6\pi}{5}, \frac{8\pi}{5}, 2\pi, \frac{12\pi}{5}, \frac{14\pi}{5}, \frac{16\pi}{5}, \frac{18\pi}{5}, 4\pi, \frac{22\pi}{5}, \frac{24\pi}{5}, \frac{26\pi}{5}, \frac{28\pi}{5}]$

Unwrapped phase data ranging over all numbers.


Range of restricted phase data $-\pi < \theta' \le \pi$
$\sin(\theta')$
$\frac{2\pi}{5}$
$\frac{4\pi}{5}$
Large jump in phase value
π
$(-\pi)$
θ'
0
$\cos(\theta')$
$\frac{-4\pi}{5}$
$\frac{-2\pi}{5}$
Range of unwrapped phase data: all numbers
$\sin(\theta)$
$\frac{2\pi}{5}, \frac{12\pi}{5}, \frac{22\pi}{5}, \ldots$
$\frac{4\pi}{5}, \frac{14\pi}{5}, \frac{24\pi}{5}, \ldots$
θ
$0, 2\pi, 4\pi, \ldots$
$\cos(\theta)$
$\frac{6\pi}{5}, \frac{16\pi}{5}, \frac{26\pi}{5}, \ldots$
$\frac{8\pi}{5}, \frac{18\pi}{5}, \frac{28\pi}{5}, \ldots$

Dialog Box



Unwrap (mask) (link)

Adds or subtracts appropriate multiples of 2pi to each input element to remove phase discontinuities (unwrap). Inputs should be radian phase values.

When the parameter "Do not unwrap phase discontinuities between successive frames" is unchecked, the block unwraps each input channel by considering all phase discontinuities in the channel, including those in previous frames of the channel. (For frame-based inputs, each column is a channel; for sample-based inputs, each element is a channel.)

When the parameter is checked, the block unwraps by considering phase discontinuities in the current frame only; the block unwraps each column of all frame-based inputs, and unwraps each column of sample-based inputs when the inputs are not row vectors. It unwraps each row of sample-based row vectors.

Parameters

□ Do not unwrap phase discontinuities between successive frames

Tolerance (radians): pi

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 selected, 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 7-783.

Tolerance

The jump size that the block recognizes as a true phase discontinuity. The default is set to π (rather than a smaller value) to avoid altering legitimate signal features. To increase the block's sensitivity, set **Tolerance** to a value slightly less than π.

Supported Data Types

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

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

See Also

`unwrap`	MATLAB

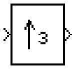
Purpose

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 $0 \leq D < (L-1)$, so that any of the L possible output phases can be selected.

This block supports triggered subsystems if, for **Frame-based mode**, you select `Maintain input frame rate`.

Sample-Based Operation

When the input is sample based, the block treats each of the M∗N matrix elements as an independent channel, and upsamples each channel over time. The **Frame-based mode** parameter must be set to **Maintain input frame size**. The output sample rate is L times higher than the input sample rate ($T_{so} = T_{si}/L$), and the input and output sizes are identical.

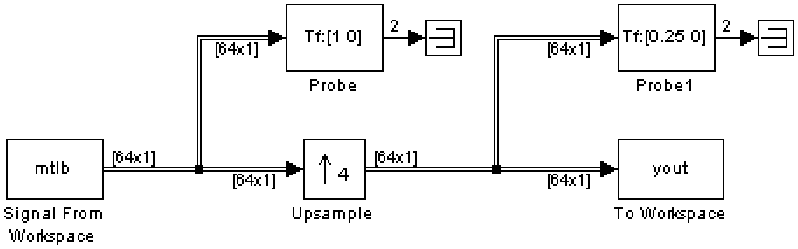
Frame-Based Operation

When the input is frame based, the block treats each of the N input columns as a frame containing M_i sequential time samples from an independent channel. The block upsamples each channel independently by inserting L-1 rows of zeros between each row in the input matrix. The **Frame-based mode** parameter determines how the block adjusts the rate at the output to accommodate the added rows. There are two available options:

- `Maintain input frame size`

  The block generates the output at the faster (upsampled) rate by using a proportionally shorter frame *period* at the output port than at the input port. For upsampling by a factor of L, the output frame period is L times shorter than the input frame period ($T_{fo} = T_{fi}/L$), but the input and output frame sizes are equal.

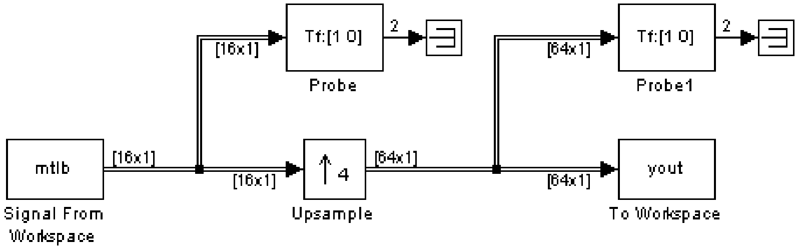
  The model below shows a single-channel input with a frame period of 1 second being upsampled by a factor of 4 to a frame period of 0.25 second. The input and output frame sizes are identical.



- `Maintain input frame rate`

  The block generates the output at the faster (upsampled) rate by using a proportionally larger frame *size* than the input. For upsampling by a factor of L, the output frame size is L times larger than the input frame size ($M_o = M_i*L$), but the input and output frame rates are equal.

  The model below shows a single-channel input of frame size 16 being upsampled by a factor of 4 to a frame size of 64. The input and output frame rates are identical.



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.

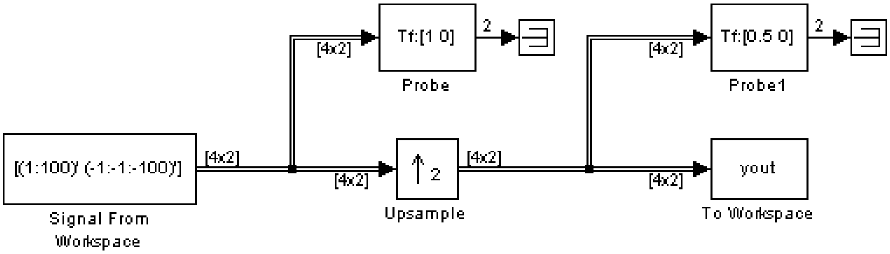
Nonzero Latency. The Upsample block has tasking latency only for multirate operation in the Simulink multitasking mode:

- 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 2-98 and the topic called 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).

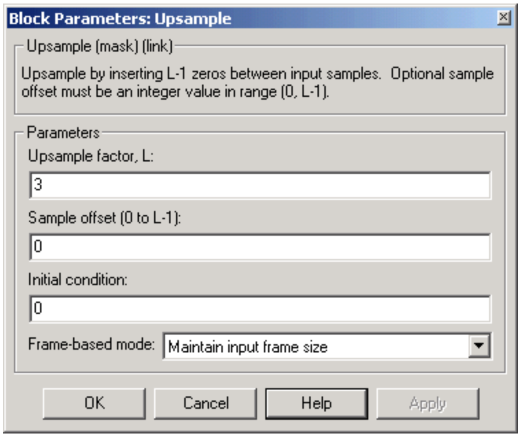
This model is multirate because there are at least two distinct frame rates, as shown by the two Probe blocks. To run this model in the Simulink multitasking mode, select `Fixed-step` and `discrete` from the **Type** controls in the **Solver** panel of the **Simulation Parameters** dialog box, and select `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 (that is, sample D+1, where D is the **Sample offset** value). It is followed by the other three initial condition rows, each separated by L-1 inserted rows of zeros, where L is the **Upsample factor** value of 2. The first row of the first input matrix appears in the output as sample 10 (that is, sample M_iL+D+1, where M_i is the input frame size).

Dialog Box



Block Parameters: Upsample

Upsample (mask) (link)

Upsample by inserting L-1 zeros between input samples. Optional sample offset must be an integer value in range [0, L-1].

Parameters

Upsample factor, L: 3

Sample offset (0 to L-1): 0

Initial condition: 0

Frame-based mode: Maintain input frame size

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` (that is, increase the frame rate), or `Maintain input frame rate` (that is, increase the frame size). The **Framing** parameter must be set to `Maintain input frame size` for sample-base inputs.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point
- Custom data types
- Boolean

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

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

See Also

Downsample	DSP Blockset
FIR Interpolation	DSP Blockset
FIR Rate Conversion	DSP Blockset
Repeat	DSP Blockset

Variable Fractional Delay

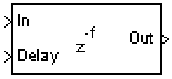
Purpose

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, can be 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. It can also be a scalar floating-point value, $0 \le v \le D$, by which to equally delay all channels.

A 1-D vector input is treated as an M_i-by-1 matrix, and the output is 1-D.

The **Initial conditions** parameter specifies the values in the block's memory at the start of the simulation in the same manner as the Variable Integer Delay block. See the Variable Integer Delay block reference page for more information.

Frame-Based Operation

When the input is frame based, the block treats each of the N input columns as a frame containing M_i sequential time samples from an independent channel.

The input to the `Delay` port, v, contains floating-point values in the range $0 \leq v \leq D$ specifying the number of sample intervals to delay the current input. The input to the `Delay` port can be

- A scalar value by which to equally delay all channels
- An M_i-by-N matrix containing the number of sample intervals to delay *each* sample in *each* channel of the current input
- An M_i-by-1 matrix containing the number of sample intervals to delay each sample in *every* channel of the current input
- A 1-by-N matrix containing the number of sample intervals to delay *every* sample in each channel of the current input

For example, if v is the M_i-by-1 matrix `[v(1) v(2) ... v(Mi)]'`, the earliest sample in the current frame is delayed by `v(1)` fractional sample intervals, the following sample in the frame is delayed by `v(2)` fractional sample intervals, and so on. The set of fractional delays contained in v is applied identically to every channel of a multichannel input.

The **Initial conditions** parameter specifies the values in the block's memory at the start of the simulation in the same manner as the Variable Integer Delay block. See the Variable Integer Delay block reference page for more information.

Interpolation Modes

The delay value specified at the `Delay` port is used as an index into the block's memory, `U`, which stores the D+1 most recent samples received at the `In` port for each channel. For example, an integer delay of `5` on a scalar input sequence retrieves and outputs the fifth most recent input sample from the block's memory, `U(6)`. Fractional delays are computed by interpolating between stored samples; the two available interpolation modes are described below.

Linear Interpolation Mode. For noninteger delays, at each sample time the **Linear Interpolation** mode uses the two samples in memory nearest to the specified delay to compute a value for the sample at that time. If `v` is the specified fractional delay for a scalar input, the output sample, `y`, is computed as follows.

```
vi = floor(v)                    % vi = integer delay
vf = v-vi                        % vf = fractional delay
y = (1-vf)*U(vi) + vf*U(vi+1)
```

Delay values less than 0 are clipped to 0, and delay values greater than D are clipped to D, where D is the **Maximum delay**. Note that a delay value of 0 causes the block to pass through the current input sample, `U(1)`, in the same simulation step that it is received.

FIR Interpolation Mode. In **FIR Interpolation** mode, the block computes a value for the sample at the desired delay by applying an FIR filter of order 2P to the stored samples on either side of the desired delay, where P is the **Interpolation filter half-length**. For periodic signals, a larger value of P (that is, a higher order filter) yields a better estimate of the sample at the specified delay. A value between 4 and 6 for this parameter (that is, a 7th to 11th order filter) is usually adequate.

A vector of 2P filter tap weights is precomputed at the start of the simulation for each of Q-1 discrete points between input samples, where 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**.

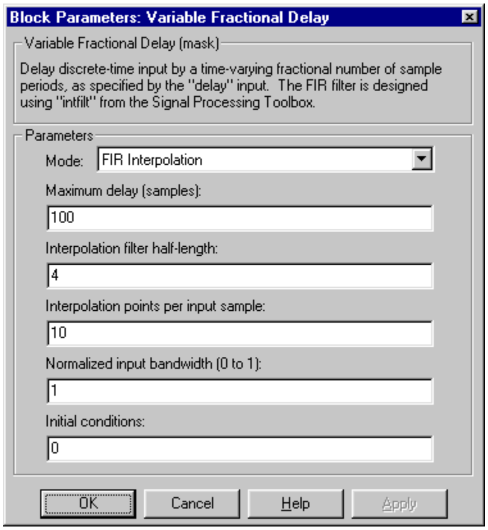
The block uses the `intfilt` function in the Signal Processing Toolbox to compute the FIR filters.

Note When the Variable Fractional Delay block is used in a feedback loop, at least one block with nonzero delay (for example, a 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



Block Parameters: Variable Fractional Delay

Variable Fractional Delay (mask)

Delay discrete-time input by a time-varying fractional number of sample periods, as specified by the "delay" input. The FIR filter is designed using "intfilt" from the Signal Processing Toolbox.

Parameters

Mode: FIR Interpolation

Maximum delay (samples): 100

Interpolation filter half-length: 4

Interpolation points per input sample: 10

Normalized input bandwidth (0 to 1): 1

Initial conditions: 0

Mode

The method by which to interpolate between adjacent stored samples to obtain a value for the sample indexed by the input at the `Delay` port.

Maximum delay

The maximum delay that the block can produce, D. Delay input values exceeding this maximum are clipped at the maximum.

Interpolation filter half-length

Half the number of input samples to use in the FIR interpolation filter.

Interpolation points per input sample

The number of points per input sample, Q, at which a unique FIR interpolation filter is computed.

Normalized input bandwidth

The bandwidth to which the interpolated output samples should be constrained. A value of 1 specifies half the sample frequency.

Initial conditions

The values with which the block's memory is initialized. See the Variable Integer Delay block for more information.

Supported Data Types

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

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

See Also

Delay	DSP Blockset
Integer Delay	DSP Blockset
Unit Delay	Simulink
Variable Integer Delay	DSP Blockset

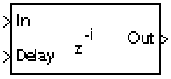
Purpose

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 Delay block in the following ways.

Variable Integer Delay Block	Delay Block
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.

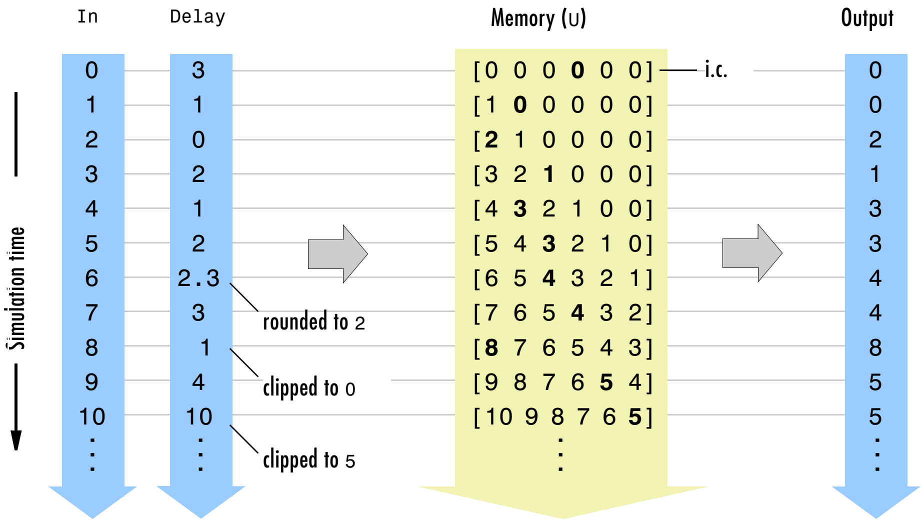
The Variable Integer Delay block stores the D+1 most recent samples received at the `In` port for each channel. At each sample time the block outputs the stored sample(s) indexed by the input to the `Delay` port.

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.



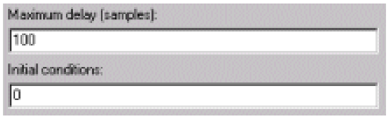
In	Delay	Memory (U)	Output
0	3	[0 0 0 **0** 0 0]	0
1	1	[1 **0** 0 0 0 0]	0
2	0	[**2** 1 0 0 0 0]	2
3	2	[3 2 **1** 0 0 0]	1
4	1	[4 **3** 2 1 0 0]	3
5	2	[5 4 **3** 2 1 0]	3
6	2.3	[6 5 **4** 3 2 1]	4
7	3	[7 6 5 **4** 3 2]	4
8	1	[**8** 7 6 5 4 3]	8
9	4	[9 8 7 6 **5** 4]	5
10	10	[10 9 8 7 6 **5**]	5
⋮	⋮	⋮	⋮

Note that the current input at each time-step is immediately stored in memory as `U(1)`. This allows the current input to be available at the output for a delay of 0 (`v`=0).

The **Initial conditions** parameter specifies the values in the block's memory at the start of the simulation. Unlike the Delay block, the Variable Integer Delay block does not have a fixed *initial delay* period during which the initial conditions appear at the output. Instead, the initial conditions are propagated to the output only when they are indexed in memory by the value at the `Delay` port. Both fixed and time-varying initial conditions can be specified in a variety of ways to suit the dimensions of the input sequence.

Fixed Initial Conditions. The settings shown below specify *fixed* initial conditions. For a fixed initial condition, the block initializes each of D samples in memory to the value entered in the **Initial conditions** parameter. A fixed initial condition in sample-based mode can be specified in one of the following ways:

- *Scalar* value with which to initialize every sample of every channel in memory. For a general M-by-N input and the parameter settings below,

  Maximum delay (samples):
  100
  Initial conditions:
  0

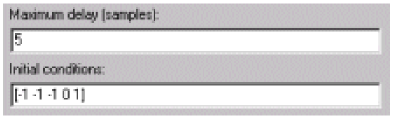
  the block initializes 100 M-by-N matrices in memory with zeros.

- *Array* of size M-by-N-by-D. In this case, you can specify different fixed initial conditions for each channel. See the *Array* bullet in "Time-Varying Initial Conditions" below for details.

Initial conditions cannot be specified by full matrices.

Time-Varying Initial Conditions. The following settings specify *time-varying* initial conditions. For a time-varying initial condition, the block initializes each of D samples in memory to one of the values entered in the **Initial conditions** parameter. This allows you to specify a unique output value for each sample in memory. A time-varying initial condition in sample-based mode can be specified in one of the following ways:

- *Vector* containing D elements with which to initialize memory samples `U(2:D+1)`, where D is the **Maximum delay**. For a scalar input and the parameters shown below, the block initializes `U(2:6)` with values `[-1, -1, -1, 0, 1]`.



Maximum delay (samples):

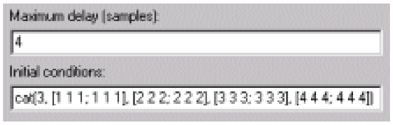
5

Initial conditions:

[-1 -1 -1 0 1]

- *Array* of dimension M-by-N-by-D with which to initialize memory samples `U(2:D+1)`, where D is the **Maximum delay** and M and N are the number of rows and columns, respectively, in the input matrix. For a 2-by-3 input and the parameters below, the block initializes memory locations `U(2:5)` with values

$$U(2) = \begin{bmatrix}1 & 1 & 1\\1 & 1 & 1\end{bmatrix}, U(3) = \begin{bmatrix}2 & 2 & 2\\2 & 2 & 2\end{bmatrix}, U(4) = \begin{bmatrix}3 & 3 & 3\\3 & 3 & 3\end{bmatrix}, U(5) = \begin{bmatrix}4 & 4 & 4\\4 & 4 & 4\end{bmatrix}$$



Maximum delay (samples):

4

Initial conditions:

cat(3, [1 1 1; 1 1 1], [2 2 2; 2 2 2], [3 3 3; 3 3 3], [4 4 4; 4 4 4])

An array initial condition can only be used with matrix inputs.

Initial conditions cannot be specified by full matrices.

Frame-Based Operation

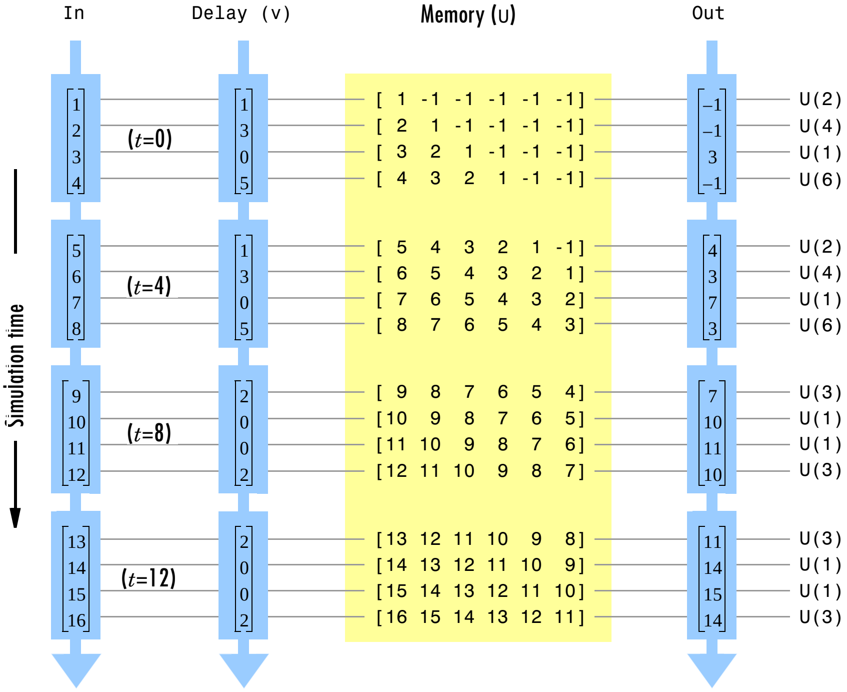
When the input is an M-by-N frame-based matrix, the block treats each of the N input columns as a frame containing M sequential time samples from an independent channel.

In frame-based mode, the input at the `Delay` port can be a scalar value to uniformly delay every sample in every channel. It can also be a length-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.

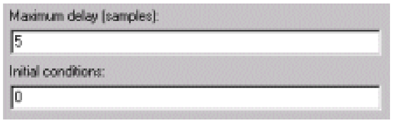
```
y(1) = U(v(1)+1)
y(2) = U(v(2)+1)
y(3) = U(v(3)+1)
y(4) = U(v(4)+1)
```



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.



Maximum delay (samples):
5
Initial conditions:
0

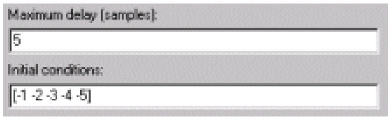
- *Array* of size 1-by-N-by-D. In this case, you can specify different fixed initial conditions for each channel. See the *Array* bullet in "Time-Varying Initial Conditions" below for details.

Initial conditions cannot be specified by full matrices.

Time-Varying Initial Conditions. The following setting specifies a *time-varying* initial condition. For a time-varying initial condition, the block initializes each of D samples in memory to one of the values entered in the **Initial conditions** parameter. This allows you to specify a unique output value for each sample in memory. A time-varying initial condition in frame-based mode can be specified in the following ways:

- *Vector* of dimensions 1-by-D. In this case, all channels have the same set of time-varying initial conditions specified by the entries of the vector. For the ramp input `[100; 100]'` with a frame size of 4, delay of 5, and the parameter settings below, the block outputs the following sequence of frames at the start of the simulation.

$$\begin{bmatrix} -1 & -1 \\ -2 & -2 \\ -3 & -3 \\ -4 & -4 \end{bmatrix}, \begin{bmatrix} -5 & -5 \\ 1 & 1 \\ 2 & 2 \\ 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 \\ 5 & 5 \\ 6 & 6 \\ 7 & 7 \end{bmatrix}, \ldots$$

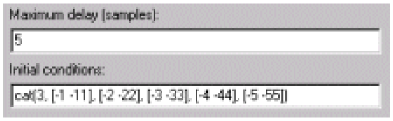


Maximum delay (samples): 5

Initial conditions: [-1 -2 -3 -4 -5]

- *Array* of size 1-by-N-by-D. In this case, you can specify different time-varying initial conditions for each channel. For the ramp input `[100; 100]'` with a frame size of 4, delay of 5, and the parameter settings below, the block outputs the following sequence of frames at the start of the simulation.

$$\begin{bmatrix} -1 & -11 \\ -2 & -22 \\ -3 & -33 \\ -4 & -44 \end{bmatrix}, \begin{bmatrix} -5 & -55 \\ 1 & 1 \\ 2 & 2 \\ 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 \\ 5 & 5 \\ 6 & 6 \\ 7 & 7 \end{bmatrix}, \ldots$$



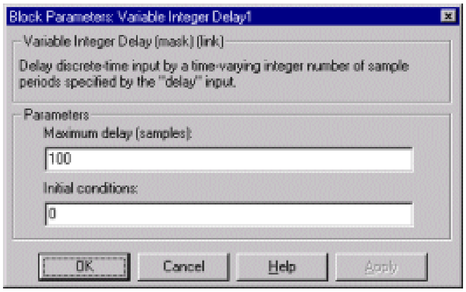
Maximum delay (samples): 5

Initial conditions: cat(3, [-1 -11], [-2 -22], [-3 -33], [-4 -44], [-5 -55])

Note that by specifying a 1-by-N-by-D initial condition array such that each 1-by-N vector entry is identical, you can implement different *fixed* initial conditions for each channel.

Initial conditions cannot be specified by full matrices.

Dialog Box



Block Parameters: Variable Integer Delay1

Variable Integer Delay (mask) (link)

Delay discrete-time input by a time-varying integer number of sample periods specified by the "delay" input.

Parameters

Maximum delay (samples): 100

Initial conditions: 0

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

See Also

Delay	DSP Blockset
Integer Delay	DSP Blockset
Variable Fractional Delay	DSP Blockset

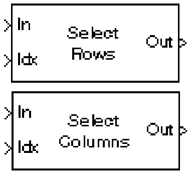
Purpose

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.

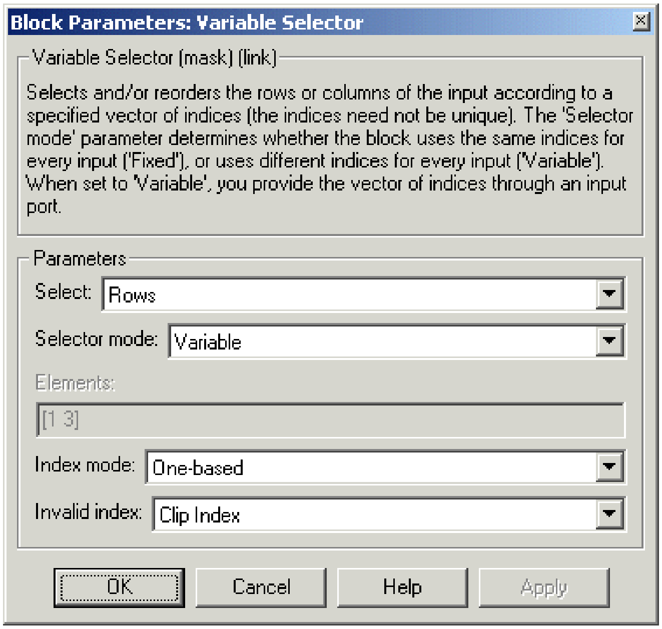
When the input is a 1-D vector, the **Select** parameter is ignored; the output is a 1-D vector of length L containing those elements specified by the length-L indexing vector.

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



Block Parameters: Variable Selector

Variable Selector (mask) (link)

Selects and/or reorders the rows or columns of the input according to a specified vector of indices (the indices need not be unique). The 'Selector mode' parameter determines whether the block uses the same indices for every input ('Fixed'), or uses different indices for every input ('Variable'). When set to 'Variable', you provide the vector of indices through an input port.

Parameters

Select: Rows

Selector mode: Variable

Elements: [1 3]

Index mode: One-based

Invalid index: Clip Index

Select

The dimension of the input to select, `Rows` or `Columns`.

Selector mode

The type of indexing operation to perform, `Variable` or `Fixed`. Variable indexing uses the input at the `Idx` port to select rows or columns from the input at the `In` port. Fixed indexing uses the **Elements** parameter value to select rows from the input at the `In` port, and disables the `Idx` port.

Elements

A vector containing the indices of the input rows or columns that will appear in the output matrix. This parameter is available when `Fixed` is selected in the **Selector mode** parameter.

Index mode

When set to `One-based`, an index value of `1` refers to the first row or column of the input. When set to `Zero-based`, an index value of `0` refers to the first row or column of the input.

Invalid index

Response to an invalid index value. Tunable.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Fixed point
- Custom data types
- Boolean
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers

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

See Also

Multiport Selector	DSP Blockset
Permute Matrix	DSP Blockset
Selector	Simulink
Submatrix	DSP Blockset

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

$$y_j = \sigma_j^2 = \frac{\sum_{i=1}^{M} \left|u_{ij} - \mu_j\right|^2}{M-1} \qquad\qquad 1 \le j \le N$$

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

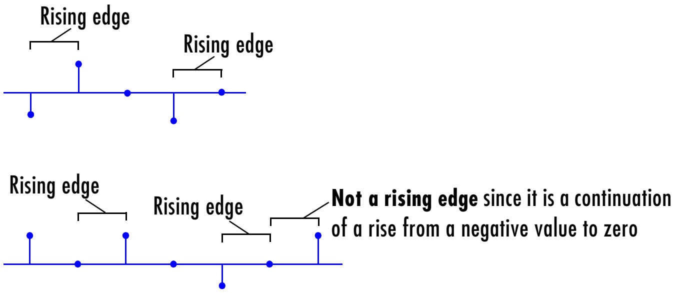
When the **Running variance** check box is selected, the block tracks the variance of each channel in a *time-sequence* of M-by-N inputs. For sample-based inputs, the output is a sample-based M-by-N matrix with each element y_{ij} containing the variance of element u_{ij} over all inputs since the last 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.

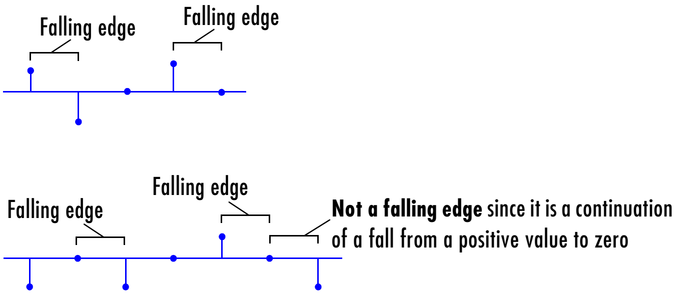
Resetting the Running Variance. The block resets the running variance whenever a reset event is detected at the optional `Rst` port. The reset signal rate must be a positive integer multiple of the rate of the data signal input.

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)


Rising edge
Rising edge
Rising edge
Rising edge
Not a rising edge since it is a continuation
of a rise from a negative value to zero

- `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)

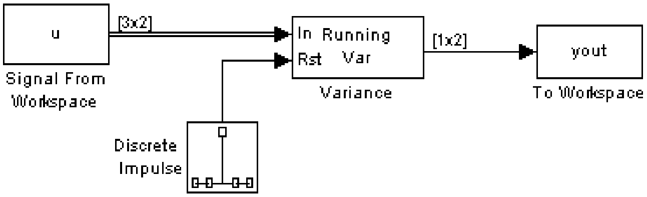

Falling edge
Falling edge
Falling edge
Falling edge
Not a falling edge since it is a continuation of a fall from a positive value to zero

- `Either edge` — Triggers a reset operation when the `Rst` input is a `Rising edge` or `Falling edge` (as described above).
- `Non-zero sample` — Triggers a reset operation at each sample time that the `Rst` input is not zero.

Note When running simulations in the Simulink `MultiTasking` mode, sample-based reset signals have a one-sample latency, and frame-based reset signals have one frame of latency. Thus, there is a one-sample or one-frame delay between the time the block detects a reset event, and when it applies the reset. For more information on latency and the Simulink tasking modes, see "Excess Algorithmic Delay (Tasking Latency)" on page 2-98 and the topic called 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`

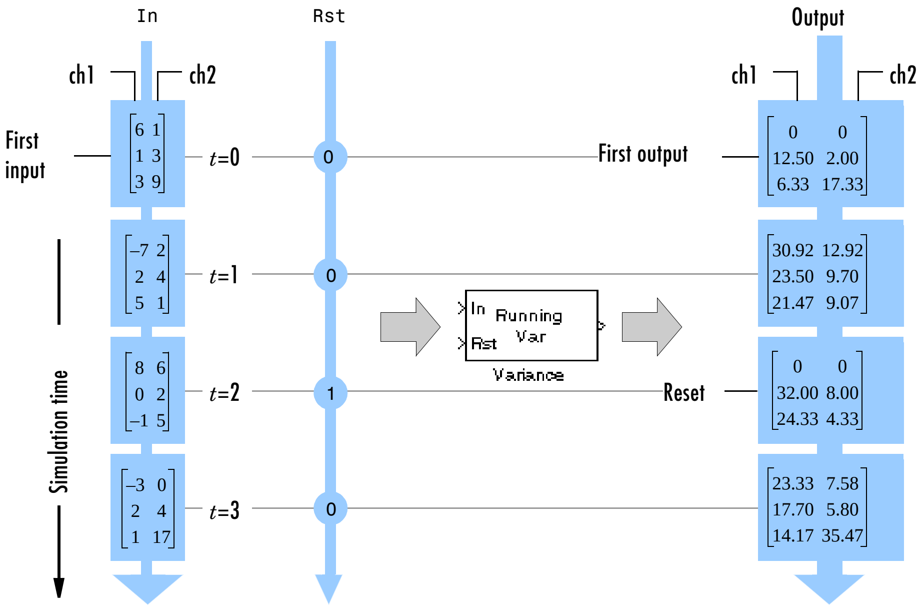
where

```
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`

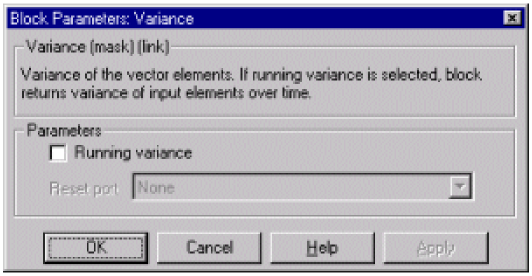
The block's operation is shown in the figure below.



In
Rst
Output
ch1
ch2
ch1
ch2
First input
$\begin{bmatrix}6 & 1\\1 & 3\\3 & 9\end{bmatrix}$
t=0
0
First output
$\begin{bmatrix}0 & 0\\12.50 & 2.00\\6.33 & 17.33\end{bmatrix}$
$\begin{bmatrix}-7 & 2\\2 & 4\\5 & 1\end{bmatrix}$
t=1
0
$\begin{bmatrix}30.92 & 12.92\\23.50 & 9.70\\21.47 & 9.07\end{bmatrix}$
In
Rst
Running Var
Variance
$\begin{bmatrix}8 & 6\\0 & 2\\-1 & 5\end{bmatrix}$
t=2
1
Reset
$\begin{bmatrix}0 & 0\\32.00 & 8.00\\24.33 & 4.33\end{bmatrix}$
$\begin{bmatrix}-3 & 0\\2 & 4\\1 & 17\end{bmatrix}$
t=3
0
$\begin{bmatrix}23.33 & 7.58\\17.70 & 5.80\\14.17 & 35.47\end{bmatrix}$
Simulation time

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

Dialog Box



Block Parameters: Variance

Variance (mask) (link)

Variance of the vector elements. If running variance is selected, block returns variance of input elements over time.

Parameters

□ Running variance

Reset port: None

Running variance

Enables running operation when selected.

Reset port

Determines the reset event that causes the block to reset the running variance. The reset signal rate must be a positive integer multiple of the rate of the data signal input. This parameter is enabled only when you select the **Running variance** check box. For more information, see “Resetting the Running Variance” on page 7-815.

Supported Data Types

- Double-precision floating point
- Single-precision floating point
- Boolean — The block accepts Boolean inputs to the `Rst` port.

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

See Also

Mean	DSP Blockset
RMS	DSP Blockset
Standard Deviation	DSP Blockset
`var`	MATLAB

Vector Scope

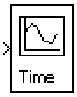
Purpose

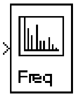
Display a vector or matrix of time-domain, frequency-domain, or user-defined data

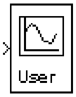
Library

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 block can be any real-valued M-by-N matrix, column or row vector, or 1-D (unoriented) vector, where 1-D vectors are treated as column vectors. Regardless of the input frame status, the block treats each column of an M-by-N input as an independent channel of data with M consecutive samples.

The block plots each sample of each input channel sequentially across the horizontal axis of the plot.

Sections of This Reference Page

Specifying the Input Domain

Select the **Show scope properties** check box to display the **Input domain** parameter. Specify the domain of the input data as `Time`, `Frequency`, or `User-defined`.

Input Domain Parameter Settings

- `Time`

Input domain: 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`

Input domain: 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)`.

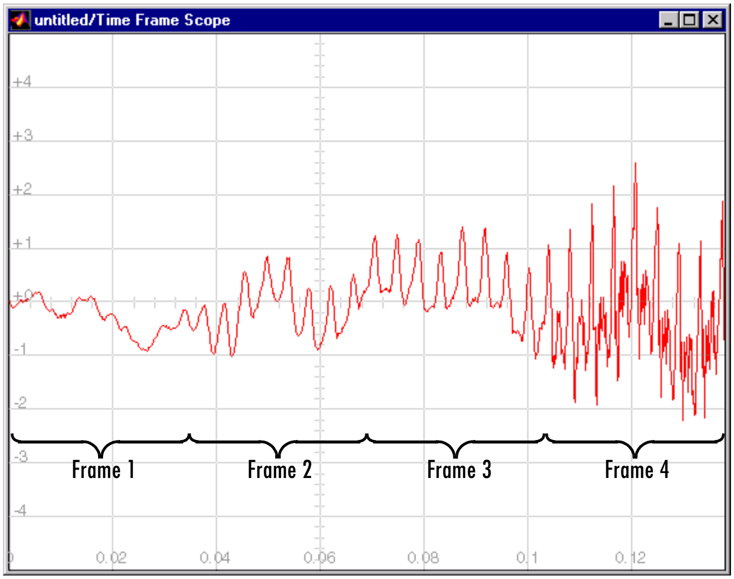
- `User-defined`

Input domain: 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 (number of frames)** parameter for time-domain signals, and the **Horizontal display span (number of frames)** parameter for user-defined signals. Setting either of these parameters 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 (Number of Frames) = 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 (number of frames)** parameter. The spacing between time points is $T_{fi}/(M-1)$, where M is the number of samples in each consecutive input frame.

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 (number of frames) 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.

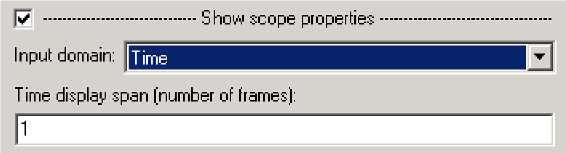
The **Frequency units** parameter specifies whether the frequency axis values should be in units of Hertz or rad/sec, and the **Frequency range** parameter specifies the range of frequencies over which the magnitudes in the input should be plotted. The available options are `[0..Fs/2]`, `[-Fs/2..Fs/2]`, and `[0..Fs]`, where `Fs` is the original time-domain signal's sample frequency.

The Vector Scope block assumes that the input data spans the range $[0,F_s)$, 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 is set to `Hertz`, the spacing between frequency points is $1/(M*T_s)$, where T_s is the sample time of the original time-domain signal. If the **Frequency units** parameter is set to `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, the **Time display span (number of frames)** 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, the **Horizontal display span (number of frames)** parameter serves the same function. Both of these parameters must be 1 or greater. See “Specifying the Input Domain” on page 7-820 for more information.



☑ Show scope properties

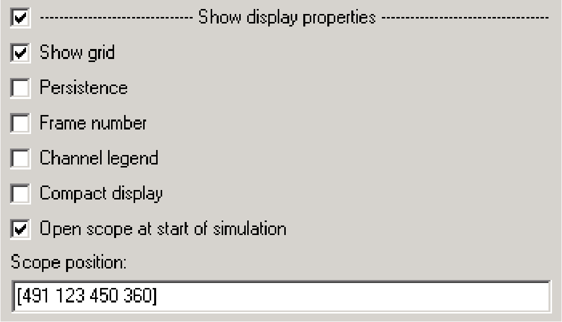
Input domain: Time

Time display span (number of frames):

1

Display Properties

The Vector Scope and Spectrum Scope blocks offer similar display property settings. You can view them by selecting the **Show display properties** check box.



☑ Show display properties

☑ Show grid

□ Persistence

□ Frame number

□ Channel legend

□ Compact display

☑ Open scope at start of simulation

Scope position:

[491 123 450 360]

The **Show grid** parameter toggles the background grid on and off.

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.

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 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.

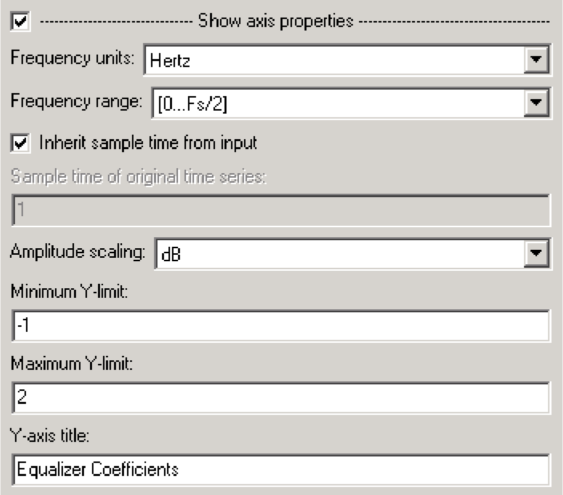
The **Scope position** parameter specifies a four-element vector of the form

```
[left bottom width height]
```

specifying the position of the scope window on the screen, where (0,0) is the lower-left corner of the display. See the MATLAB `figure` function for more information.

Axis Properties

The Vector Scope and Spectrum Scope blocks also share similar axis property settings. For the Vector Scope block, the parameters listed under the **Show axis properties** check box vary with the domain of the input. The dialog box below shows the parameters available for frequency-domain data.



☑ Show axis properties

Frequency units: Hertz

Frequency range: [0...Fs/2]

☑ Inherit sample time from input

Sample time of original time series: 1

Amplitude scaling: dB

Minimum Y-limit: -1

Maximum Y-limit: 2

Y-axis title: Equalizer Coefficients

Minimum Y-limit and **Maximum Y-limit** set the range of the vertical axis. If `Autoscale` is selected from the 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.

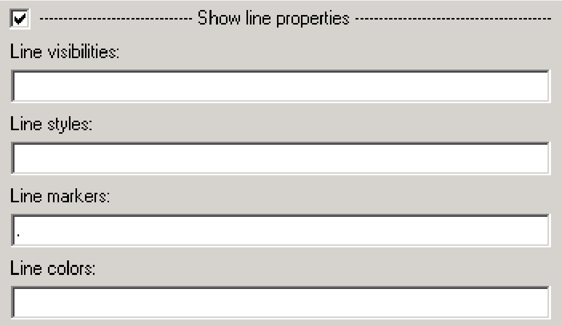
Frequency-domain and user-defined data need extra information to scale the horizontal axis. For user-defined data, the parameters that provide this information are **Inherit sample increment from input** and **Increment per sample in input frame**. See “Scaling the Horizontal Axis for User-Defined Signals” on page 7-821 for more information. For frequency-domain data, you must specify an analogous pair of parameters, **Inherit sample time from input** and **Sample time of original time series**. See “Scaling the Horizontal Axis for Frequency-Domain Signals” on page 7-822 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 described in “Scaling the Horizontal Axis for Frequency-Domain Signals” on page 7-822.

Line Properties

Both the Vector Scope and Spectrum Scope blocks also offer similar line property settings. You can view them by selecting the **Show line properties** check box.



☑ ------------------------------ Show line properties --

Line visibilities:

Line styles:

Line markers: .

Line colors:

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 channel’s data is displayed on the scope, and which is hidden. The syntax specifies the visibilities in list form, where the term `on` or `off` as a list entry specifies the visibility of the corresponding channel’s data. The list entries are separated by the pipe symbol, `|`.

A five-channel signal would ordinarily generate five distinct plots on the scope. To disable plotting of the third and fifth lines, enter the following visibility specification in the **Line visibilities** parameter.

```
  on | on | off | on | off
ch 1  ch 2  ch 3  ch 4  ch 5
```

Note that the first (leftmost) list item corresponds to the first signal channel (leftmost column of the input matrix).

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

```
   -  |  --  |  :  |  -.  |  -
ch 1   ch 2  ch 3  ch 4   ch 5
```

These settings plot the signal channels with the following styles.

Line Style	Command to Type in Line Style Parameter	Appearance
Solid	`-`	────────────
Dashed	`--`	— — — — — — — —
Dotted	`:`	································
Dash-dot	`-.`	— · — · — · — · — · — · — · — ·
No line	`none`	No line appears

Note that the first (leftmost) list item, `'-'`, corresponds to the first signal channel (leftmost column of the input matrix). See the `LineStyle` property of the MATLAB `line` function for more information about the style syntax. To specify a marker for the individual sample points, use the **Line markers** parameter, described below.

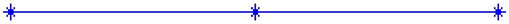
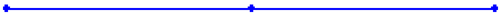
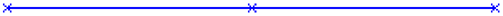
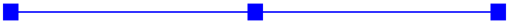
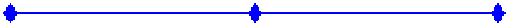
Line markers — The **Line markers** parameter specifies the marker style with which each channel's samples are represented on the scope. The syntax specifies the channels' marker styles in list form, with each list entry specifying a marker for the corresponding channel's data. The list entries are separated by the pipe symbol, `|`.

For example, a five-channel signal would ordinarily generate all five plots with no marker symbol (that is, the individual sample points are not marked on the scope). To instead plot each line with a different marker style, you could enter

```
  * | . | x | s | d
```

ch 1 ch 2 ch 3 ch 4 ch 5

These settings plot the signal channels with the following styles.

Marker Style	Command to Type in Marker Style Parameter	Appearance
Asterisk	`*`	
Point	`.`	
Cross	`x`	
Square	`s`	
Diamond	`d`	

Note that the leftmost list item, `'*'`, corresponds to the first signal channel or leftmost column of the input matrix. See the `Marker` property of the MATLAB `line` function for more information about the available markers.

To produce a *stem plot* for the data in a particular channel, type the word `stem` instead of one of the basic marker shapes.

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]
```

ch 1 ch 2 ch 3 ch 4 ch 5

or

```
  'k' | 'b' | 'r' | 'g' | [.7529 0 .7529]
 ch 1   ch 2   ch 3   ch 4          ch 5
```

These settings plot the signal channels in the following colors (8-bit RGB equivalents shown in the center column).

Color	RGB Equivalent	Appearance
Black	(0,0,0)	———
Blue	(0,0,255)	———
Red	(255,0,0)	———
Green	(0,255,0)	———
Dark purple	(192,0,192)	———

Note that the leftmost list item, `'k'`, corresponds to the first signal channel or leftmost column of the input matrix. See `ColorSpec` in the MATLAB documentation for more information about the color syntax.

Scope Window

The scope title in the window title bar is the same as the block title. In addition to the standard MATLAB figure window menus such as **File**, **Window**, and **Help**, the Vector Scope window contains **Axes** and **Channels** menus.

The parameters that you set using the **Axes** menu apply to all channels. Many of the parameters in this menu are also accessible through the **Block Parameters** dialog box. For descriptions of these parameters, see "Display Properties" on page 7-824. Below are descriptions of other parameters in the **Axes** menu:

- **Refresh** erases all data on the scope display, except for the most recent trace. This command is useful in conjunction with the **Persistence** setting.
- **Autoscale** resizes the *y*-axis to best fit the vertical range of the data. The numerical limits selected by the autoscale feature are displayed in the

Minimum Y-limit and **Maximum Y-limit** parameters in the parameter dialog box. You can 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.

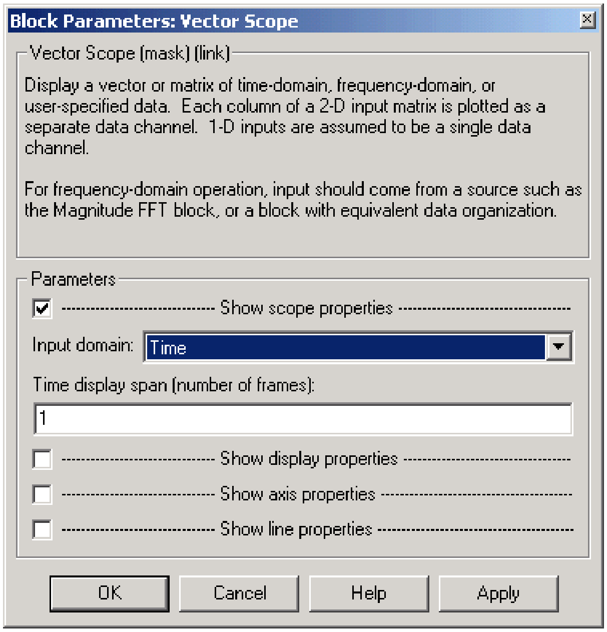
The properties listed in the **Channels** menu apply to a particular channel. All of the parameters in this menu are also accessible through the **Block Parameters** dialog box. For descriptions of these parameters, see "Line Properties" on page 7-827.

Many of these options can also be accessed by right-clicking with the mouse anywhere on the scope display. The menu that is displayed contains a combination of the options available in both the **Axes** and **Channels** menus. The right-click menu is very helpful when the scope is in zoomed mode, when the **Axes** and **Channels** menus are not visible.

Note If you select **Compact display** from the **Axes** menu, the **Axes** and **Channels** menus are no longer visible. Right-click in the Vector Scope window and click **Compact display** in order to make the menus reappear.

Dialog Box

Scope Properties Pane



Block Parameters: Vector Scope

Vector Scope (mask) (link)

Display a vector or matrix of time-domain, frequency-domain, or user-specified data. Each column of a 2-D input matrix is plotted as a separate data channel. 1-D inputs are assumed to be a single data channel.

For frequency-domain operation, input should come from a source such as the Magnitude FFT block, or a block with equivalent data organization.

Parameters

☑ ------------------------------ Show scope properties ----------------------------------

Input domain: Time

Time display span (number of frames):

1

□ ------------------------------ Show display properties ----------------------------------

□ ------------------------------ Show axis properties ----------------------------------

□ ------------------------------ Show line properties ----------------------------------

Show scope properties

Select this check box to expose the **Scope properties** pane. See “Scope Properties” on page 7-823. Tunable.

Input domain

Select the domain of the input. Your choices are `Time`, `Frequency`, or `User-defined`. See “Specifying the Input Domain” on page 7-820. Tunable.

Time display span (number of frames)

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 7-820.

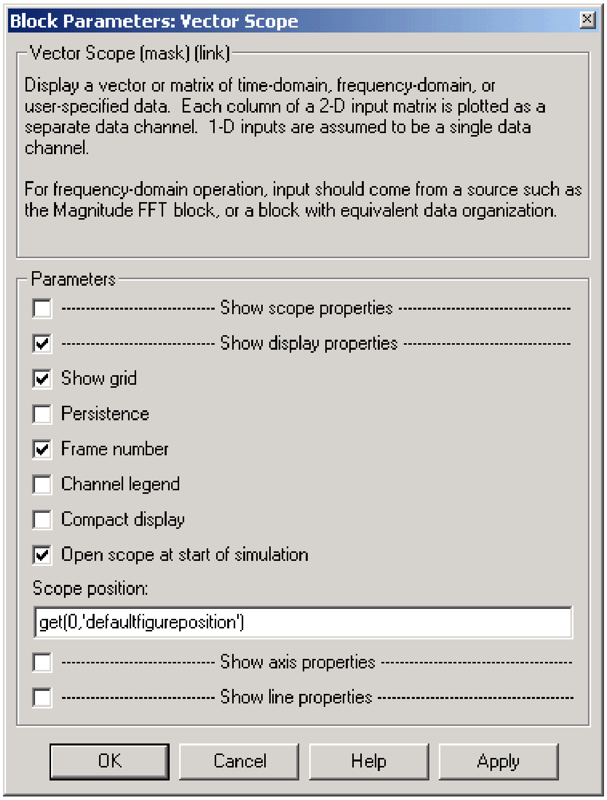
This parameter is visible when you select the **Show scope properties** check box and set the **Input domain** parameter to `Time`.

Horizontal display span (number of frames)

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 7-820.

This parameter is visible when you select the **Show scope properties** check box and set the **Input domain** parameter to `User-defined`.

Display Properties Pane



Block Parameters: Vector Scope

Vector Scope (mask) (link)

Display a vector or matrix of time-domain, frequency-domain, or user-specified data. Each column of a 2-D input matrix is plotted as a separate data channel. 1-D inputs are assumed to be a single data channel.

For frequency-domain operation, input should come from a source such as the Magnitude FFT block, or a block with equivalent data organization.

Parameters

□ ------------------------------ Show scope properties ------------------------------

☑ ------------------------------ Show display properties ------------------------------

☑ Show grid

□ Persistence

☑ Frame number

□ Channel legend

□ Compact display

☑ Open scope at start of simulation

Scope position:

get(0,'defaultfigureposition')

□ ------------------------------ Show axis properties ------------------------------

□ ------------------------------ Show line properties ------------------------------

Show display properties

Select this check box to expose **Display properties** pane. See "Display Properties" on page 7-824. Tunable.

Show grid

Toggle the scope grid on and off. See “Display Properties” on page 7-824. 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 7-824. Tunable.

Frame number

When selected, the number of the current frame in the input sequence appears in the Vector Scope window. See “Display Properties” on page 7-824. Tunable.

Channel legend

Toggles the legend on and off. See “Display Properties” on page 7-824. Tunable.

Compact display

Resizes the scope to fill the window. See “Display Properties” on page 7-824. Tunable.

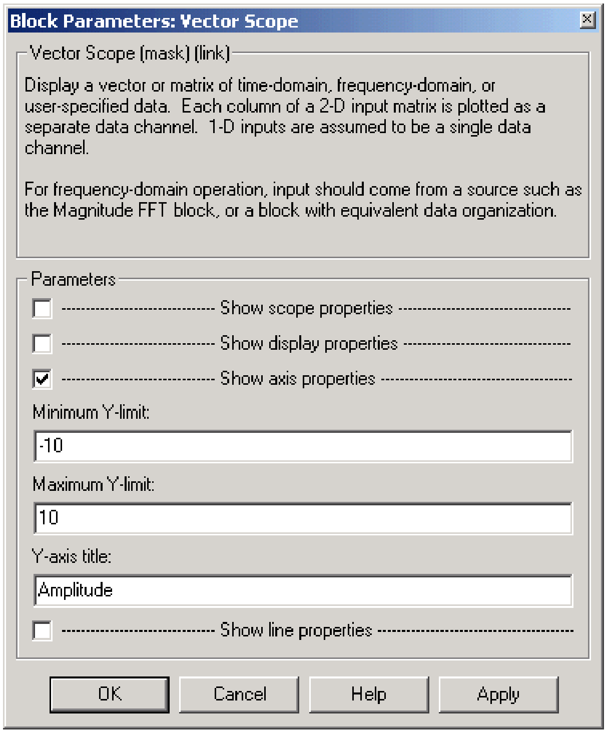
Open scope at start of simulation

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. The Vector Scope window and the **Block Parameters** dialog box open. See “Display Properties” on page 7-824. 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 7-824. Tunable.

Axis Properties Pane



Block Parameters: Vector Scope

Vector Scope (mask) (link)

Display a vector or matrix of time-domain, frequency-domain, or user-specified data. Each column of a 2-D input matrix is plotted as a separate data channel. 1-D inputs are assumed to be a single data channel.

For frequency-domain operation, input should come from a source such as the Magnitude FFT block, or a block with equivalent data organization.

Parameters

□ ---------------------------- Show scope properties ----------------------------

□ ---------------------------- Show display properties ----------------------------

☑ ---------------------------- Show axis properties ----------------------------

Minimum Y-limit: -10

Maximum Y-limit: 10

Y-axis title: Amplitude

□ ---------------------------- Show line properties ----------------------------

Show axis properties

Select this check box to expose the **Axis Properties** pane. See "Axis Properties" on page 7-825. 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.

Frequency units

The frequency units for the *x*-axis, `Hertz` or `rad/sec`. See “Axis Properties” on page 7-825. Tunable.

This parameter is visible when, in the **Show scope properties** pane, for **Input domain** parameter, you select `Frequency`.

Frequency range

The frequency range over which to plot the data, `[0..Fs/2]`, `[-Fs/2..Fs/2]`, or `[0..Fs]`, where `Fs` is the sample frequency of the original time-domain signal, $1/T_s$. See “Axis Properties” on page 7-825. Tunable.

This parameter is visible when, in the **Show scope properties** pane, for **Input domain** parameter, you select `Frequency`.

Inherit sample time from input

Computes the time-domain sample period from the frame period and frame size of the frequency-domain input. Use this parameter 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 it was generated. See “Axis Properties” on page 7-825. Tunable.

This parameter is visible when, in the **Show scope properties** pane, for **Input domain** parameter, you select `Frequency`.

Sample time of original time series

The sample period of the original time-domain signal, T_s. See “Axis Properties” on page 7-825. Tunable.

This parameter is visible when, in the **Show scope properties** pane, for **Input domain** parameter, you select `Frequency`. Then, select the **Show axis properties** check box and clear the **Inherit sample time from input** check box.

Amplitude scaling

The scaling for the *y*-axis, `dB` or `Magnitude`. See “Axis Properties” on page 7-825. Tunable.

This parameter is visible when, in the **Show scope properties** pane, for **Input domain** parameter, you select `Frequency`.

Inherit sample increment from input

Scales the horizontal axis by computing the horizontal interval between samples in the input frame from the frame period of the input. Use this parameter only if the input's sample period is equal to the period with which the physical signal was originally sampled. See "Axis Properties" on page 7-825. Tunable.

This parameter is visible when, in the **Show scope properties** pane, for **Input domain** parameter, you select `User-defined`.

Increment per sample in input frame

The numerical interval between adjacent *x*-axis points corresponding to the user-defined input data. See "Axis Properties" on page 7-825. Tunable.

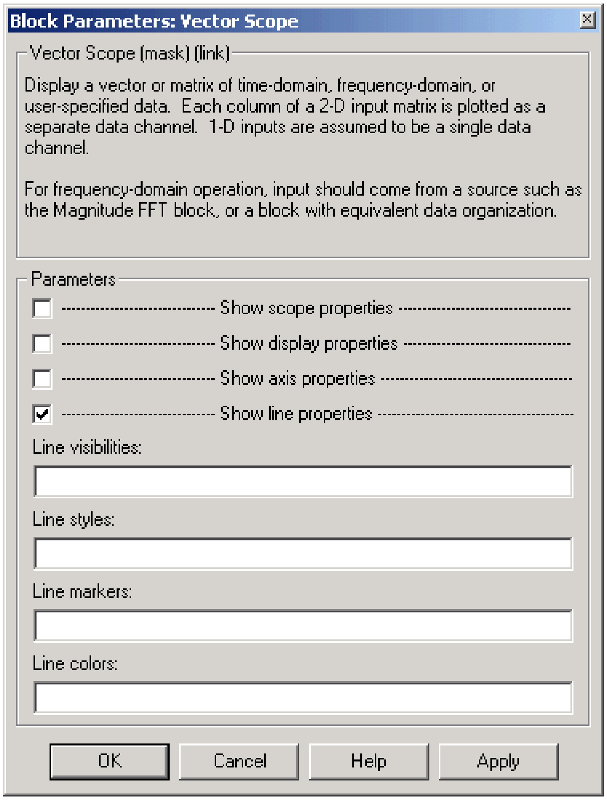
This parameter is visible when, in the **Show scope properties** pane, for **Input domain** parameter, you select `User-defined`. Then, select the **Show axis properties** check box and clear the **Inherit sample increment from input** check box.

X-axis title

The text to be displayed below the *x*-axis. Tunable.

This parameter is visible when, in the **Show scope properties** pane, for **Input domain** parameter, you select `User-defined`.

Line Properties Pane


Block Parameters: Vector Scope

Vector Scope (mask) (link)

Display a vector or matrix of time-domain, frequency-domain, or user-specified data. Each column of a 2-D input matrix is plotted as a separate data channel. 1-D inputs are assumed to be a single data channel.

For frequency-domain operation, input should come from a source such as the Magnitude FFT block, or a block with equivalent data organization.

Parameters

□ ------------------------------ Show scope properties ------------------------------

□ ------------------------------ Show display properties ------------------------------

□ ------------------------------ Show axis properties ------------------------------

☑ ------------------------------ Show line properties ------------------------------

Line visibilities:

Line styles:

Line markers:

Line colors:

Show line properties

Select this check box to expose the **Line Properties** pane. See “Line Properties” on page 7-827. 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 7-827. Tunable.

Line styles

The line styles of the various channels’ scope traces. Channels are separated by a pipe (|) symbol. See “Line Properties” on page 7-827. Tunable.

Line markers

The line markers of the various channels' scope traces. Channels are separated by a pipe (`|`) symbol. See "Line Properties" on page 7-827. 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 7-827. 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 your data types to the above data types in MATLAB and Simulink, see "Supported Data Types and How to Convert to Them" on page A-2.

See Also

Matrix Viewer	DSP Blockset
Spectrum Scope	DSP Blockset

Also see "Viewing Signals" on page 2-87 for how to use this and other blocks to view signals.

Purpose

Compute a window, and/or apply a window to an input signal

Library

Signal Operations

Description

hamming
In
Out

hamming
Win

hamming
In
Out
Win

The Window Function block computes a window, and/or applies a window to an input signal. This block supports real and complex floating-point and fixed-point inputs.

Operation Modes

The Window Function block has three modes of operation that you can select via the **Operation** parameter. In each mode, the block first creates a window vector w by sampling the window specified in the **Window type** parameter at M discrete points. The operation modes are

- `Apply window to input`

  In this mode, the block computes an M-by-1 window vector w and multiplies it element-wise with each of the N channels in the M-by-N input matrix u. This is equivalent to the following MATLAB code:

  ```
  y = repmat(w,1,N) .* u         % Equivalent MATLAB code
  ```

  In this mode, a length-M 1-D vector input is treated as an M-by-1 matrix. The output y always has the same dimension as the input. 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 for this mode.

- `Generate and apply window`

  In this mode, the block computes an M-by-1 window vector w and multiplies it element-wise with each of the N channels in the M-by-N input matrix u. This is equivalent to the following MATLAB code:

  ```
  y = repmat(w,1,N) .* u         % Equivalent MATLAB code
  ```

  In this mode, a length-M 1-D vector input is treated as an M-by-1 matrix. The block produces two outputs:

  - At the Out port, the block produces the result of the multiplication y, which has the same dimension as the input. If the input is frame based, the output y is frame based; otherwise, the output y is sample based.

- At the Win port, the block produces the M-by-1 window vector w. Output w is always sample based.

Window Type

The available window types are shown in the table below. For complete information about the window functions, consult the Signal Processing Toolbox documentation.

Window Type	Description
Bartlett	Computes a Bartlett window. `w = bartlett(M)`
Blackman	Computes a Blackman window. `w = blackman(M)`
Boxcar	Computes a rectangular window. `w = rectwin(M)`
Chebyshev	Computes a Chebyshev window with stopband ripple `R`. `w = chebwin(M,R)`
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)`

Window Type	Description
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,x1,...,xn) % Window takes extra parameters {x1 ... xn}`

Window Sampling

For the generalized-cosine windows (`Blackman`, `Hamming`, `Hann`, and `Hanning`), the **Sampling** parameter determines whether the window samples are computed in a *periodic* or a *symmetric* manner. For example, if **Sampling** is set to `Symmetric`, a Hamming window of length M is computed as

```
w = hamming(M)     % Symmetric (aperiodic) window
```

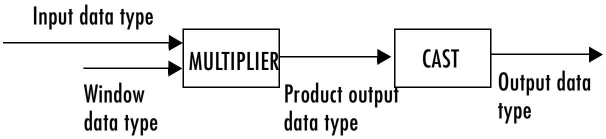
If **Sampling** is set to `Periodic`, the same window is computed as

```
w = hamming(M+1)   % Periodic (asymmetric) window
w = w(1:M)
```

Fixed-Point Data Types

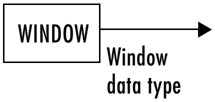
The following diagram shows the data types used within the Window block for fixed-point signals for each of the three operating modes.

Apply window to input



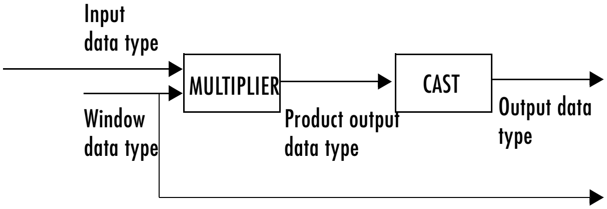
The input data type comes from the driving block. You can set the window, product output, and output data types in the block mask. In this mode, the window vector is not output from the block.

Generate window



In this mode, the block acts as a source. The window vector is output in the window data type you specify in the block n

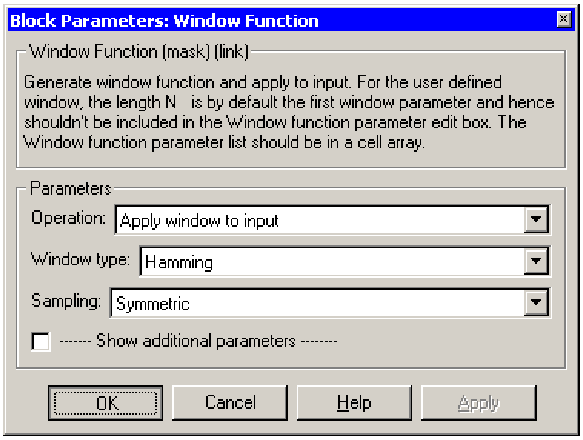
Generate and apply window



The input data type comes from the driving block. You can set the window, product output, and output data types in the block mask. In this mode, the window vector is output from the block.

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

Dialog Box



Block Parameters: Window Function

Window Function (mask) (link)

Generate window function and apply to input. For the user defined window, the length N   is by default the first window parameter and hence shouldn't be included in the Window function parameter edit box. The Window function parameter list should be in a cell array.

Parameters

Operation: Apply window to input

Window type: Hamming

Sampling: Symmetric

□ ------ Show additional parameters -------

Operation

Specify the block's operation as discussed in "Operation Modes" on page 7-840. The port configuration of the block is updated to match the setting of this parameter.

Window type

Specify the type of window to apply as listed in "Window Type" on page 7-841. This is a tunable parameter.

Sample mode

(Not shown in dialog above.) Specify the sample mode for the block, `Continuous` or `Discrete`, when it is in `Generate Window` mode. In the `Apply window to output` and `Generate and apply window` modes, the block inherits the sample time from its driving block. Therefore, this parameter is only visible when `Generate window` is selected for the **Operation** parameter.

Sample time

(Not shown in dialog above.) Specify the sample time for the block when it is in `Generate window` and `Discrete` modes. In `Apply window to output` and `Generate and apply window` modes, the block inherits the sample time from its driving block. This parameter is only visible when `Discrete` is selected for the **Sample mode** parameter.

Window length

(Not shown in dialog above.) Specify the length of the window to apply. This parameter is only visible when `Generate window` is selected for the **Operation** parameter. Otherwise, the window vector length is computed to match the input frame size, M.

Sampling

Specify the window sampling for generalized-cosine windows. This parameter is only visible if `Blackman`, `Hamming`, `Hann`, or `Hanning` is selected for the **Window type** parameter. This is a tunable parameter.

Stopband attenuation in dB

(Not shown in dialog above.) Specify the level of stopband attenuation, R_s, in decibels. This parameter is only visible if `Chebyshev` is selected for the **Window type** parameter. This is a tunable parameter.

Beta

(Not shown in dialog above.) Specify the `Kaiser` window β parameter. Increasing β widens the mainlobe and decreases the amplitude of the window sidelobes in the window's frequency magnitude response. This parameter is only visible if `Kaiser` is selected for the **Window type** parameter. This parameter is tunable.

Window function name

(Not shown in dialog above.) Specify the name of the user-defined window function to be calculated by the block. This parameter is only visible if `User defined` is selected for the **Window type** parameter.

Additional parameters for user defined window

(Not shown in dialog above.) Select to enable the **Window function parameters** parameter, when the user-defined window requires parameters other than the window length. This parameter is only visible if `User defined` is selected for the **Window type** parameter.

Window function parameters

(Not shown in dialog above.) Specify the extra parameters required by the user-defined window function, besides the window length. This parameter is only available when the **Additional parameters for user defined window** parameter is selected. The entry must be a cell array.

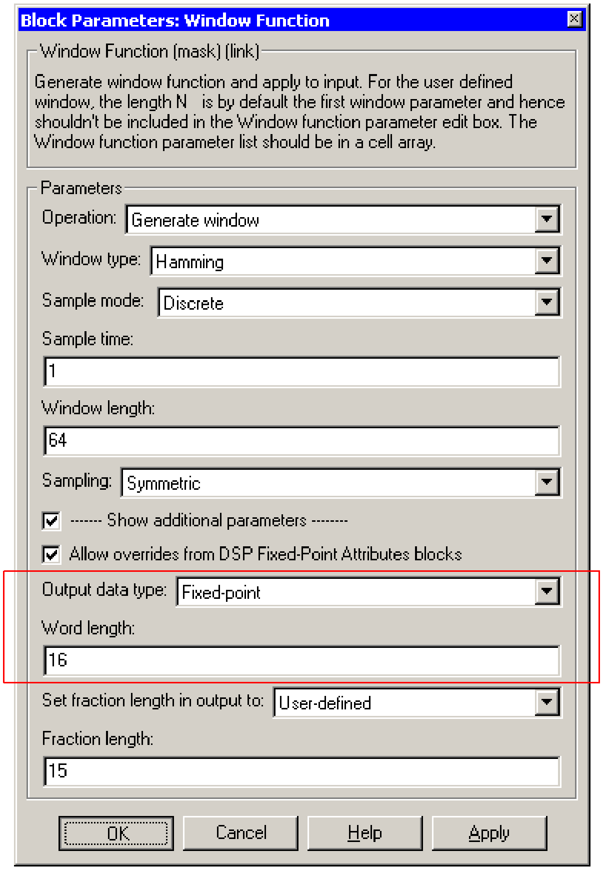
Show additional parameters

If selected, additional parameters specific to implementation of the block become visible as shown in the sections below:

- “Parameters for Generate Window Only Mode” on page 7-846
- “Parameters for Apply Window Modes” on page 7-848

Parameters for Generate Window Only Mode

The following parameters are available when the **Operation** parameter is set to `Generate window`:



Block Parameters: Window Function

Window Function (mask) (link)

Generate window function and apply to input. For the user defined window, the length N is by default the first window parameter and hence shouldn't be included in the Window function parameter edit box. The Window function parameter list should be in a cell array.

Parameters

Operation: Generate window

Window type: Hamming

Sample mode: Discrete

Sample time: 1

Window length: 64

Sampling: Symmetric

☑ ------ Show additional parameters -------

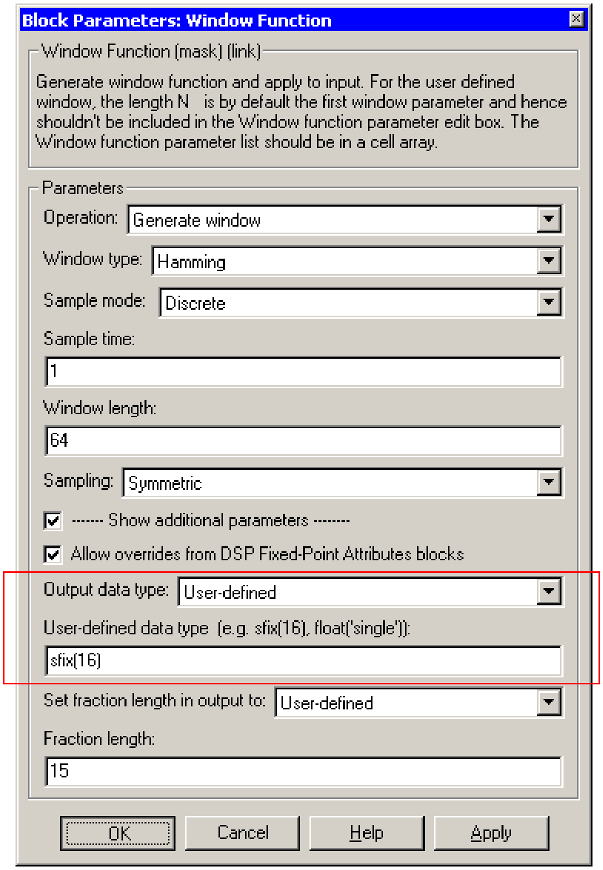
☑ Allow overrides from DSP Fixed-Point Attributes blocks

Output data type: Fixed-point

Word length: 16

Set fraction length in output to: User-defined

Fraction length: 15



Block Parameters: Window Function

Window Function (mask) (link)

Generate window function and apply to input. For the user defined window, the length N is by default the first window parameter and hence shouldn't be included in the Window function parameter edit box. The Window function parameter list should be in a cell array.

Parameters

Operation: Generate window

Window type: Hamming

Sample mode: Discrete

Sample time: 1

Window length: 64

Sampling: Symmetric

☑ ------ Show additional parameters -------

☑ Allow overrides from DSP Fixed-Point Attributes blocks

Output data type: User-defined

User-defined data type (e.g. sfix(16), float('single')): sfix(16)

Set fraction length in output to: User-defined

Fraction length: 15

Allow overrides from DSP Fixed-Point Attributes blocks

If you select this parameter, fixed-point data types for this block may be set by DSP Fixed-Point Attributes blocks in your model. If this parameter is unselected, the data types are always set by the parameters in the block mask.

Output data type

Specify the output data type in one of the following ways:

- Choose `double` or `single` from the drop-down list
- Choose `Fixed-point` to specify the output data type and scaling in the **Word length**, **Set fraction length in output to,** and **Fraction length** parameters
- Choose `User-defined` to specify the output data type and scaling in the **User-defined data type**, **Set fraction length in output to,** and **Fraction length** parameters
- Choose `Inherit via back propagation` to set the output data type and scaling to match the following block

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 fraction length in output to

Specify the scaling of the fixed-point output by either of the following two methods:

- Choose `Best precision` to have the output scaling automatically set such that the output signal has the best possible precision.
- Choose `User-defined` to specify the output scaling in the **Fraction length** parameter.

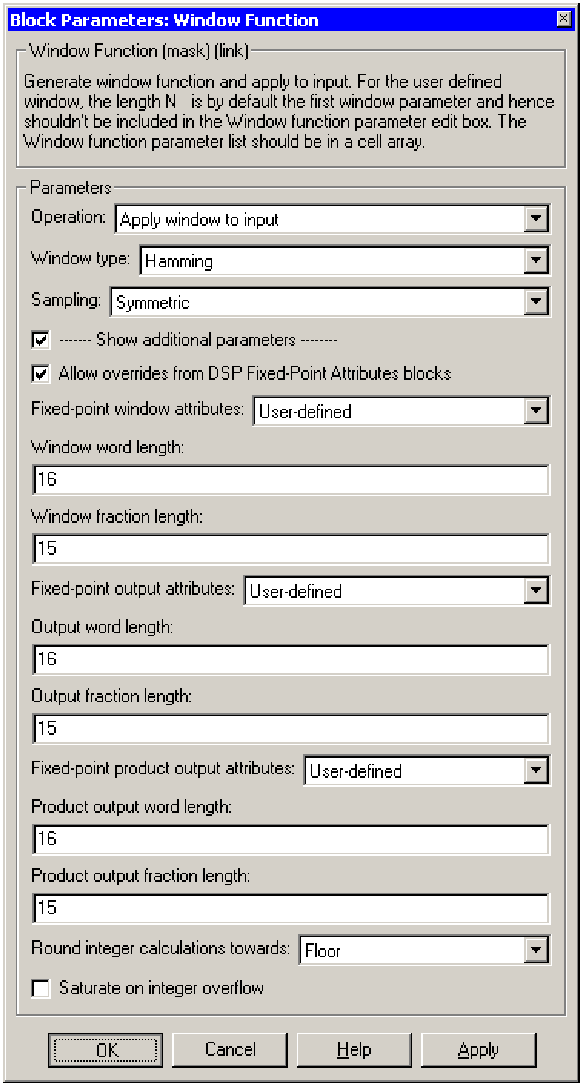
This parameter is only visible if `Fixed-point` or `User-defined` is selected for the **Output data type** parameter, and if the specified output data type is a fixed-point data type.

Fraction length

Specify the fraction length, in bits, of the fixed-point output data type. 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 fraction length in output to** parameter.

Parameters for Apply Window Modes

The following parameters are available when the **Operation** parameter is set to either `Apply window to input` or `Generate and apply window`.



Block Parameters: Window Function

Window Function (mask) (link)

Generate window function and apply to input. For the user defined window, the length N   is by default the first window parameter and hence shouldn't be included in the Window function parameter edit box. The Window function parameter list should be in a cell array.

Parameters

Operation: Apply window to input

Window type: Hamming

Sampling: Symmetric

☑ ------ Show additional parameters -------

☑ Allow overrides from DSP Fixed-Point Attributes blocks

Fixed-point window attributes: User-defined

Window word length: 16

Window fraction length: 15

Fixed-point output attributes: User-defined

Output word length: 16

Output fraction length: 15

Fixed-point product output attributes: User-defined

Product output word length: 16

Product output fraction length: 15

Round integer calculations towards: Floor

☐ Saturate on integer overflow

Allow overrides from DSP Fixed-Point Attributes blocks

If you select this parameter, fixed-point data types for this block may be set by DSP Fixed-Point Attributes blocks in your model. If this parameter is unselected, the data types are always set by the parameters in the block mask.

Fixed-point window attributes

Choose how you will specify the word length and fraction length of the window vector `w`. If you select `Same as input`, these characteristics will match those of the input to the block. If you select `User-defined`, the **Window word length** and **Window fraction length** parameters become visible.

The window vector does not obey the **Round integer calculations toward** and **Saturate on integer overflow** parameters; it is always saturated and rounded to `Nearest`.

Window word length

Specify the word length, in bits, of the data type and scaling of the window vector, `w`. This parameter is only visible when `User-defined` is specified for the **Fixed-point window attributes** parameter.

Window fraction length

Specify the fraction length, in bits, of the data type and scaling of the window vector, `w`. This parameter is only visible when `User-defined` is specified for the **Fixed-point window attributes** parameter.

Fixed-point output attributes

Choose how you will specify the output word length and fraction length. If you select `Same as input`, these characteristics will match those of the input to the block. If you select `User-defined`, the **Output word length** and **Output fraction length** parameters become visible.

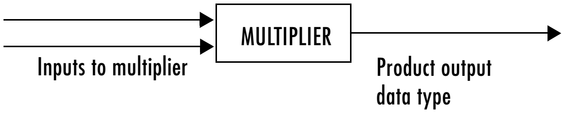
Output word length

Specify the word length, in bits, of the output. This parameter is only visible if `User-defined` is specified for the **Fixed-point output attributes** parameter.

Output fraction length

Specify the fraction length, in bits, of the output. This parameter is only visible if `User-defined` is specified for the **Fixed-point output attributes** parameter.

Fixed-point product output attributes



As depicted above, the output of the multiplier is placed into the product output data type and scaling. Use this parameter to specify how you would like to designate this product output word and fraction lengths.

If you select `Same as output`, the product output word and fraction lengths are the same as those of the output of the block. If you select `User-defined`, the **Product output word length** and **Product output fraction length** parameters become visible.

Product output word length

Specify the word length, in bits, of the product output. This parameter is only visible when `User-defined` is specified for the **Fixed-point product output attributes** parameter.

Product output fraction length

Specify the fraction length, in bits, of the product output. This parameter is only visible when `User-defined` is specified for the **Fixed-point product output attributes** parameter.

Round integer calculations toward

Select the rounding mode for fixed-point operations.

The window vector `w` does not obey this parameter; it always rounds to `Nearest`.

Saturate on integer overflow

If selected, overflows saturate.

The window vector `w` does not obey this parameter; it is always saturated.

Supported Data Types

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

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

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

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

Yule-Walker AR Estimator

Purpose

Compute an estimate of AR model parameters using the Yule-Walker method

Library

Estimation / Parametric Estimation

Description

Yule AR
Estimator
A
G

Yule AR
Estimator
K
G

Yule AR
Estimator
A
K
G

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.

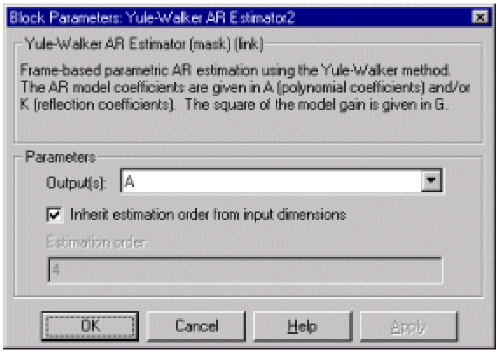
When **Output(s)** is set to `A`, port A is enabled. Port A outputs a column vector of length p+1 that contains the normalized estimate of the AR model coefficients in descending powers of z

```
[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



Block Parameters: Yule-Walker AR Estimator2

Yule-Walker AR Estimator (mask) (link)

Frame-based parametric AR estimation using the Yule-Walker method. The AR model coefficients are given in A (polynomial coefficients) and/or K (reflection coefficients). The square of the model gain is given in G.

Parameters

Output(s): A

☑ Inherit estimation order from input dimensions

Estimation order

4

Output(s)

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

Inherit estimation order from input dimensions

When selected, sets the estimation order *p* to one less than the length of the input vector. Nontunable.

Estimation order

The order of the AR model, *p*. This parameter is enabled when **Inherit estimation order from input dimensions** is not selected. Nontunable.

References

Kay, S. M. *Modern Spectral Estimation: Theory and Application.* Englewood Cliffs, NJ: Prentice-Hall, 1988.

Marple, S. L., Jr., *Digital Spectral Analysis with Applications.* Englewood Cliffs, NJ: Prentice-Hall, 1987.

Supported Data Types

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

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

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

Yule-Walker Method

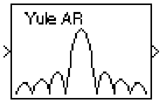
Purpose

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.

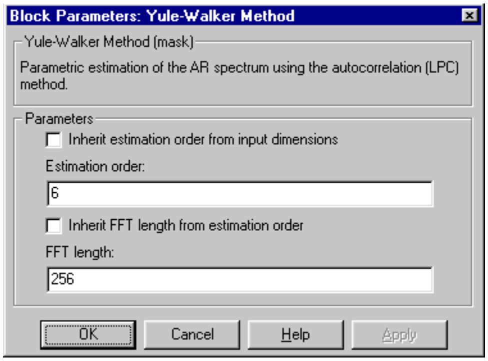
The input is a sample-based vector (row, column, or 1-D) or frame-based vector (column only) representing a frame of consecutive time samples from a single-channel signal. The block's output (a column vector) is the estimate of the signal's power spectral density at N_{fft} equally spaced frequency points 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



Block Parameters: Yule-Walker Method

Yule-Walker Method (mask)

Parametric estimation of the AR spectrum using the autocorrelation (LPC) method.

Parameters

□ Inherit estimation order from input dimensions

Estimation order: 6

□ Inherit FFT length from estimation order

FFT length: 256

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 your data types to the above data types in MATLAB and Simulink, see “Supported Data Types and How to Convert to Them” on page A-2.

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 5-5 for related information.

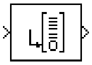
Purpose

Alter the input dimensions by zero-padding (or truncating) rows and/or columns

Library

Signal Operations

Description



The Zero Pad block changes the dimensions of the input matrix from M_i-by-N_i to M_o-by-N_o by zero-padding or truncating along the columns, rows, or columns and rows. Use the **Pad along** parameter to specify the dimensions to change.

Using the **Pad signal at** parameter, you can choose to pad your input matrix at the end or the beginning of a row and/or column.

The **Number of output rows** and/or **Number of output columns** parameters refer to the dimensions of the output, M_o and N_o. You can set these parameters to `User-specified` or `Next power of two`. If you choose `User-specified`, enter a scalar value in the **Specified number of output rows** and/or **Specified number of output columns** parameters. If you choose `Next power of two`, the block pads the input matrix along the columns and/or rows until the length of the columns and/or rows is equal to a power of two. If the length of the input matrix's columns and/or rows is already equal to a power of two, the block does not pad the input matrix.

If you choose `User-specified` for the **Number of output rows** and/or **Number of output columns** parameters, you can specify a scalar value in the **Specified number of output rows** and/or **Specified number of output columns** parameters that truncates the size of your input matrix. The following options are available for the **Action when truncation occurs** parameter:

- `None` — Select this option if you do not want to be notified that the input matrix is truncated.
- `Warning` — Choose this option if you want a warning to be displayed in the MATLAB Command Window when the input matrix is truncated.
- `Error` — Click this option if you want an error dialog box to be displayed and the simulation terminated when the input matrix is truncated.

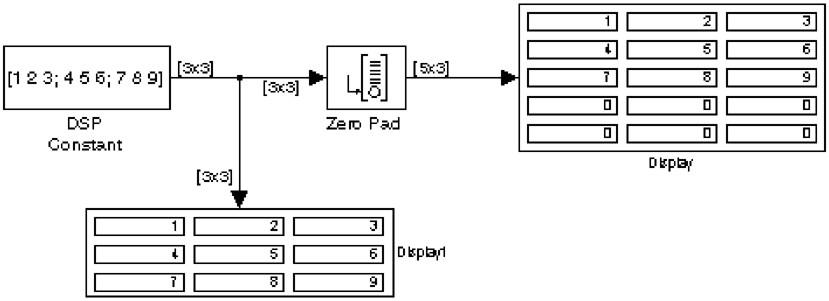
The behavior of the Pad block and Zero Pad block is identical, with the exception that the Pad block can pad the input matrix with values other than zero. See the Pad block reference page for more information on the behavior of the Pad block.

Example

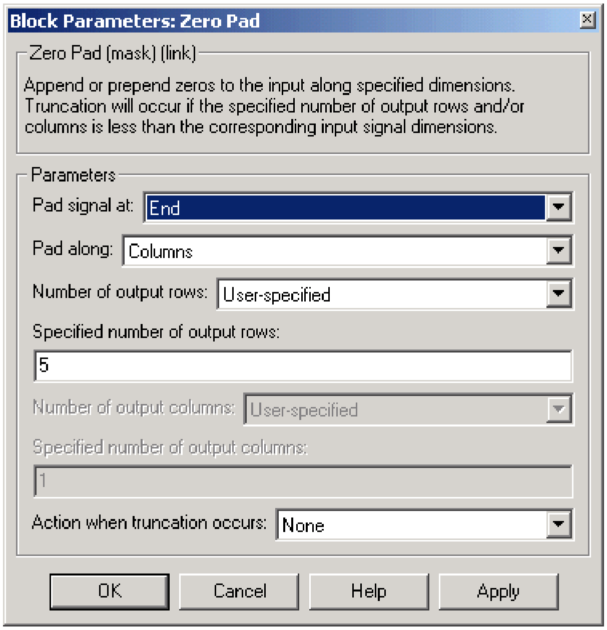
In the model below, the 3-by-3 input is zero-padded along the column dimension to 5-by-3. The parameter settings in the Zero Pad block are

- **Pad signal at:** `End`
- **Pad along:** `Columns`
- **Number of output rows:** `User-specified`
- **Specified number of output rows** = `5`
- **Action when truncation occurs:** `None`

The following figure shows the result of running the model.



Dialog Box



Block Parameters: Zero Pad

Zero Pad (mask) (link)

Append or prepend zeros to the input along specified dimensions. Truncation will occur if the specified number of output rows and/or columns is less than the corresponding input signal dimensions.

Parameters

Pad signal at: End

Pad along: Columns

Number of output rows: User-specified

Specified number of output rows: 5

Number of output columns: User-specified

Specified number of output columns: 1

Action when truncation occurs: None

Pad signal at

The input matrix can be padded at the beginning of the rows and/or columns or at the end of the rows and/or columns.

Pad along

The direction along which to pad or truncate. `Columns` specifies that the *row* dimension should be changed to M_o. `Rows` specifies that the *column* dimension should be changed to N_o. `Columns and rows` specifies that both column and row dimensions should be changed. `None` disables padding and truncation and passes the input through to the output unchanged.

Number of output rows

The total number of output rows can be `User-specified` or `Next power of two`. If you select `User-specified`, type a scalar value in the **Specified Number of output rows** parameter. If you select `Next power of two`, the block pads the columns of the input matrix until the number of rows is equal to a power of two. If the number of rows is already equal to a power of two, the block does not pad the input matrix.

Specified number of output rows

The desired number of rows in the output, M_0. This parameter is enabled when `Columns` or `Columns and rows` is selected in the **Pad along** menu and `User-specified` is chosen in the **Number of output rows** parameter.

Number of output columns

The total number of output columns. This parameter is enabled when `Rows` or `Columns and rows` is selected in the **Pad along** menu. If you select `User-specified`, type a scalar value in the **Specified Number of output columns** parameter. If you select `Next power of two`, the block pads the rows of the input matrix until the number of columns is equal to a power of two. If the number of columns is already equal to a power of two, the block does not pad the input matrix.

Specified number of output columns

The desired number of columns in the output, N_0. This parameter is enabled when `Rows` or `Columns and rows` is selected in the **Pad along** menu and `User-specified` is chosen in the **Number of output columns** parameter.

Action when truncation occurs

Choose `None` if you do not want to be notified that the input matrix is truncated. Select `Warning` to display a warning when the input matrix is truncated. Choose `Error` if you want an error dialog box to be displayed and the simulation terminated when the input matrix is truncated.

Supported Data Types

- Double-precision floating-point
- Single-precision floating-point
- Fixed point
- Boolean
- 8-, 16-, and 32-bit signed integers
- 8-, 16-, and 32-bit unsigned integers
- Custom data types

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

See Also

Matrix Concatenation	Simulink
Pad	DSP Blockset
Repeat	DSP Blockset
Submatrix	DSP Blockset
Upsample	DSP Blockset
Variable Selector	DSP Blockset

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
SingleTaskRate TransMsg	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 **Output buffer size** parameter setting, and `m` is the **Buffer overlap** parameter setting.

The blocks whose delay can be computed by `rebuffer_delay` are

- Buffer
- Unbuffer

`d = rebuffer_delay(f,n,m,'singletasking')` returns the delay (in samples) introduced by these blocks in single-tasking operations.

The table below shows the appropriate `rebuffer_delay` parameter values to use in computing delay for the two blocks.

Block	Parameter Values
Buffer	`f` = input frame size (`f=1` for sample-based mode) `n` = **Output buffer size** `m` = **Buffer overlap**

Unbuffer	`f` = input frame size
	`n = 1`
	`m = 0`

See Also

Buffer	DSP Blockset
Unbuffer	DSP Blockset

Data Type Support

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

Supported Data Types and How to Convert to Them

Note All data type support applies to both simulation and Real-Time Workshop C code generation. All 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

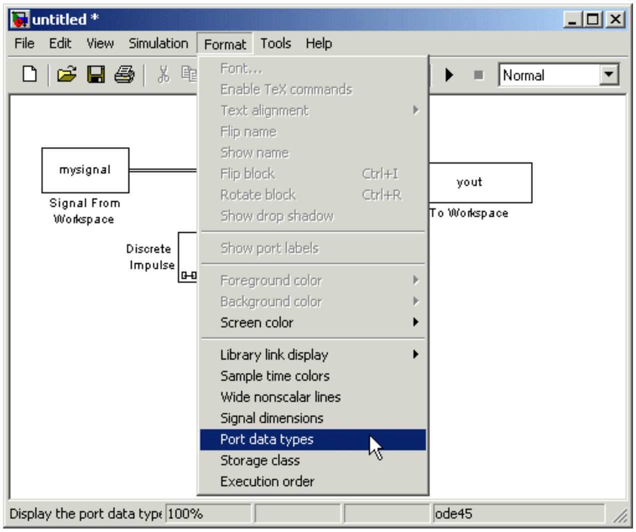
Data Types Supported by DSP Blockset Blocks	Commands and Blocks for Converting Data Types	Comments
Double-precision floating point	• `double` • Data Type Conversion block	Simulink built-in data type supported by all DSP Blockset blocks.
Single-precision floating point	• `single` • Data Type Conversion 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-6.
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)

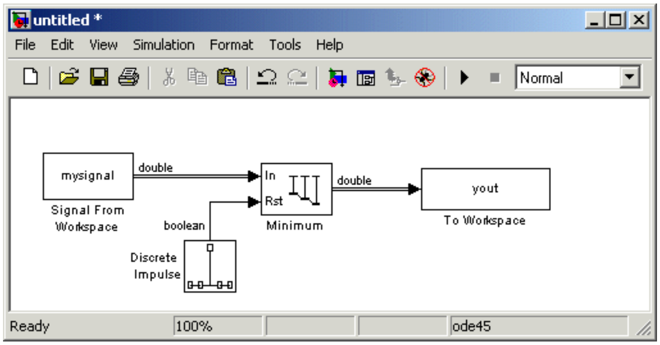
Data Types Supported by DSP Blockset Blocks	Commands and Blocks for Converting Data Types	Comments
Fixed-point data types	• Blocks in the Data Type library of the Fixed-Point Blockset • Fixed-Point Blockset functions for conversions (listed in the online categorical function reference of Fixed-Point 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 data types in the DSP Blockset, see Chapter 6, "Fixed-Point Support."
Custom data types	See "Correctly Defining Custom Data Types" on page A-5 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 below. The signal lines in the model will then have labels indicating their data types as shown below. (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.

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-6
- "Effects of Enabling and Disabling Boolean Support" on page A-7
- "Steps to Disabling Boolean Support" on page A-8

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

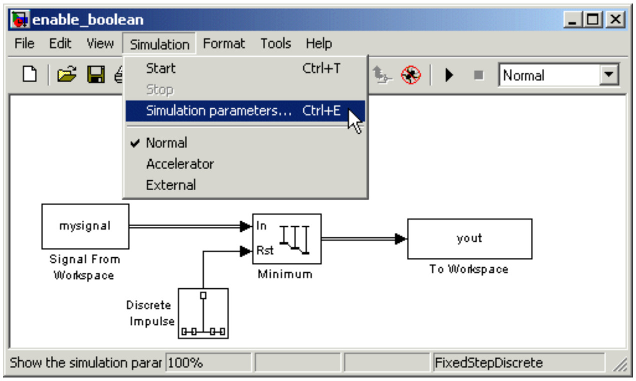
To disable Boolean data type support in a particular model, clear the Boolean-enabling simulation parameter in the model by completing the following:

- "Step 1: Open the Simulation Parameters Dialog Box" on page A-9
- "Step 2: Disable the Boolean Data Type in the Advanced Tab" on page A-10
- "Step 3: (Optional) Verify Data Types of Signals" on page A-10

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.

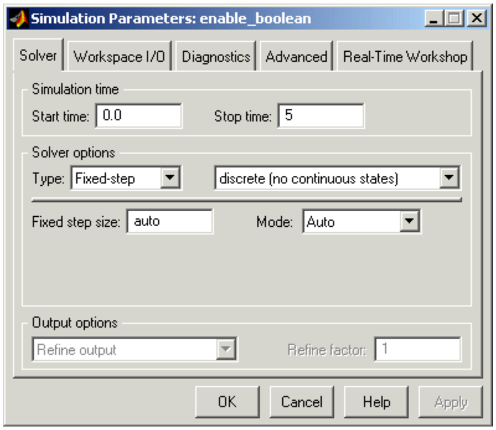
Step 1: Open the Simulation Parameters Dialog Box

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...**



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).



Simulation time

Start time: 0.0   Stop time: 5

Solver options

Type: Fixed-step   discrete (no continuous states)

Fixed step size: auto   Mode: Auto

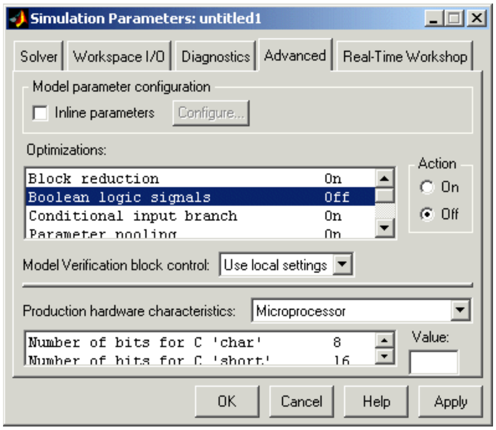
Output options

Refine output   Refine factor: 1

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**. 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-7.



Model parameter configuration

□ Inline parameters

Optimizations:

```
Block reduction              On
Boolean logic signals        Off
Conditional input branch     On
Parameter pooling            On
```

Action

○ On

● Off

Model Verification block control: Use local settings

Production hardware characteristics: Microprocessor

```
Number of bits for C 'char'     8
Number of bits for C 'short'    16
```

Value:

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-4.) 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-7).

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

You can generate ANSI/ISO C code from DSP Blockset blocks by using Real-Time Workshop or Real-Time Workshop Embedded Coder.

ANSI/ISO C Code Generation Support

You can generate ANSI/ISO C code from DSP Blockset blocks by using Real-Time Workshop 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/ISO 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/ISO 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/ISO C functions that implement core algorithms such as FFT and convolution. Run-time functions are precompiled into ANSI/ISO C *run-time libraries*, and enable the blocks to generate smaller, faster code that requires less RAM.
- **Parameter reuse (Real-Time Workshop 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/ISO 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

This appendix describes how to adjust certain Simulink settings to suit your needs.

When you create a new DSP model, you might 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

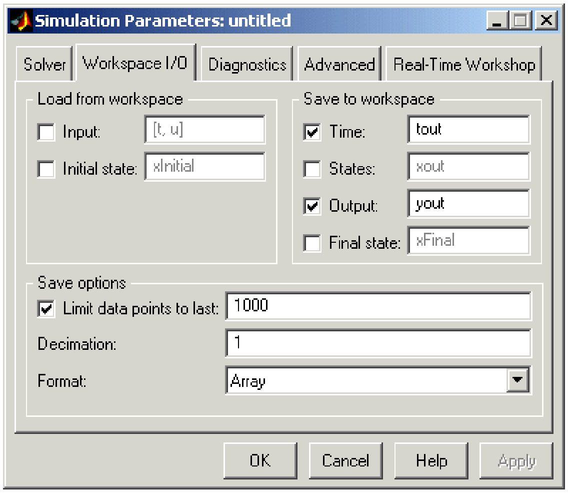
You can edit the dspstartup M-file to change any of the settings above or to add your own custom settings. For example, you can change the 'StopTime' option to a value that is better suited to your particular simulations, or set the 'SaveTime' option to 'on' if you prefer to record the simulation sample 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).



Load from workspace

□ Input: [t, u]

□ Initial state: xInitial

Save to workspace

☑ Time: tout

□ States: xout

☑ Output: yout

□ Final state: xFinal

Save options

☑ Limit data points to last: 1000

Decimation: 1

Format: Array

- `'SaveOutput'` is set to `'off'`

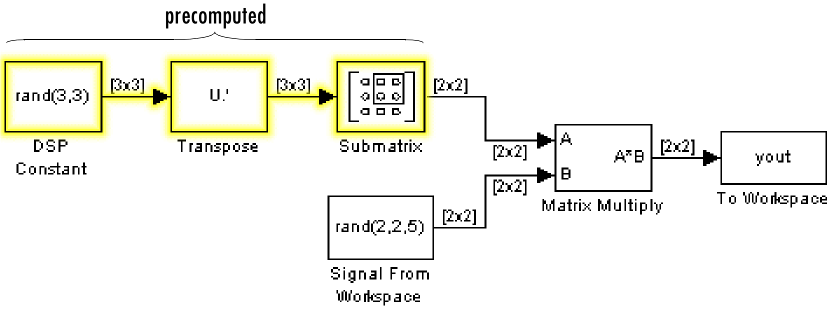
  When `'SaveOutput'` is set to `'off'`, Simulink Outport blocks in the top level of a model do not generate an output (`yout`) in the workspace. To reenable output recording for a particular model, select the **Output** check box in the **Workspace I/O** panel of the **Simulation Parameters** dialog box (above).

- `'InvariantConstants'` is set to `'on'`

  When `'InvariantConstants'` is set to `'on'`, Simulink precomputes the values of all constant blocks (for example, DSP Constant and Constant

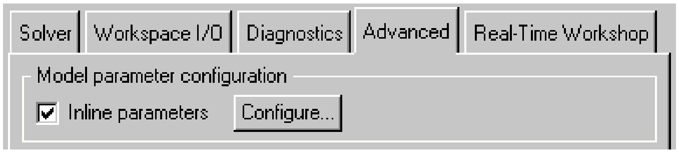
Diagonal Matrix) at the start of the simulation, and does not update them again for the duration of the simulation. Simulink additionally precomputes the outputs of all downstream blocks driven exclusively by constant blocks.

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.



Model parameter configuration

☑ Inline parameters

- `'RTWOptions'` sets loop-rolling threshold to `2`

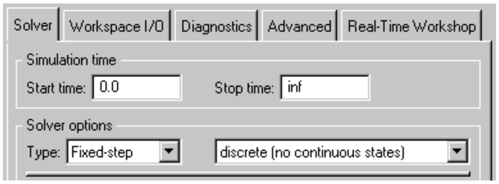
  By default, the Real-Time Workshop "unrolls" a given loop into inline code when the number of loop iterations is less than five. This avoids the overhead of servicing the loop in cases when inline code can be used with only a modest increase in the file size.

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.



Simulation time

Start time: 0.0

Stop time: inf

Solver options

Type: Fixed-step

discrete (no continuous states)

- '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 2-2 for more information about the various solver settings.

D

Glossary of Fixed-Point Terms

This glossary defines terms related to fixed-point data types and numbers. These terms may appear in some or all of the documents that describe products from The MathWorks that have fixed-point support.

arithmetic shift

A shift of the bits of a binary word for which the sign bit is recycled for each bit shift to the right. A zero is incorporated into the least significant bit of the word for each bit shift to the left. In the absence of overflows, each arithmetic shift to the right is equivalent to a division by 2, and each arithmetic shift to the left is equivalent to a multiplication by 2.

See Also binary point, binary word, bit, logical shift, most significant bit

bias

Part of the numerical representation used to interpret a fixed-point number. Along with the slope, the bias forms the scaling of the number. Fixed-point numbers can be represented as

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

where the slope can be expressed as

$$\textit{slope} = \textit{fractional slope} \times 2^{\textit{exponent}}$$

See Also fixed-point representation, fractional slope, integer, scaling, slope, [Slope Bias]

binary number

A value represented in a system of numbers that has two as its base and that uses 1's and 0's (bits) for its notation.

See Also bit

binary point

A symbol in the shape of a period that separates the integer and fractional parts of a binary number. Bits to the left of the binary point are integer bits and/or sign bits, and bits to the right of the binary point are fractional bits.

See Also binary number, bit, fraction, integer, radix point

binary point-only scaling

The scaling of a binary number that results from shifting the binary point of the number right or left, and which therefore can only occur by powers of two.

See Also binary number, binary point, scaling

binary word

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

See Also bit, data type, word

bit

The smallest unit of information in computer software or hardware. A bit can have the value 0 or 1.

ceiling (round toward)

A rounding mode that rounds to the closest representable number in the direction of positive infinity.

See Also floor (round toward), nearest (round toward), rounding, truncation, zero (round toward)

contiguous binary point

A binary point that occurs within the word length of a data type. For example, if a data type has four bits, its contiguous binary point must be understood to occur at one of the following five positions:

.0000
0.000
00.00
000.0
0000.

See Also data type, noncontiguous binary point, word length

data type

A set of characteristics that define a group of values. A fixed-point data type is defined by its word length, its fraction length, and whether it is signed or unsigned. A floating-point data type is defined by its word length and whether it is signed or unsigned.

See Also fixed-point representation, floating-point representation, fraction length, word length

data type override

A parameter in the **Fixed-Point Settings** interface that allows you to set the output data type and scaling of Fixed-Point Blockset blocks on a system or subsystem level.

See Also data type, scaling

exponent

Part of the numerical representation used to express a floating-point or fixed-point number.

1. Floating-point numbers are typically represented as

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

2. Fixed-point numbers can be represented as

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

where the slope can be expressed as

$$\mathit{slope} = \mathit{fractional\,slope} \times 2^{\mathit{exponent}}$$

The exponent of a fixed-point number is equal to the negative of the fraction length:

$$\mathit{exponent} = -1 \times \mathit{fraction\ length}$$

See Also bias, fixed-point representation, floating-point representation, fraction length, fractional slope, integer, mantissa, slope

fixed-point representation

A method for representing numerical values and data types that have a set range and precision.

1. Fixed-point numbers can be represented as

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

where the slope can be expressed as

$$\textit{slope} = \textit{fractional slope} \times 2^{\textit{exponent}}$$

The slope and the bias together represent the scaling of the fixed-point number.

2. Fixed-point data types can be defined by their word length, their fraction length, and whether they are signed or unsigned.

See Also bias, data type, exponent, fraction length, fractional slope, integer, precision, range, scaling, slope, word length

floating-point representation

A method for representing numerical values and data types that can have changing range and precision.

1. Floating-point numbers can be represented as

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

2. Floating-point data types are defined by their word length.

See Also data type, exponent, mantissa, precision, range, word length

floor (round toward)

A rounding mode that rounds to the closest representable number in the direction of negative infinity.

See Also ceiling (round toward), nearest (round toward), rounding, truncation, zero (round toward)

fraction

The part of a fixed-point number represented by the bits to the right of the binary point. The fraction represents numbers that are less than one.

See Also binary point, bit, fixed-point representation

fraction length

The number of bits to the right of the binary point in a fixed-point representation of a number.

See Also binary point, bit, fixed-point representation, fraction

fractional slope

Part of the numerical representation used to express a fixed-point number. Fixed-point numbers can be represented as

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

where the slope can be expressed as

$$\textit{slope} = \textit{fractional slope} \times 2^{\textit{exponent}}$$

See Also bias, exponent, fixed-point representation, integer, slope

guard bits

Extra bits in either a hardware register or software simulation that are added to the high end of a binary word to ensure that no information is lost in case of overflow.

See Also binary word, bit, overflow

integer

1. The part of a fixed-point number represented by the bits to the left of the binary point. The integer represents numbers that are greater than or equal to one.

2. Also called the “stored integer.” The raw binary number, in which the binary point assumed to be at the far right of the word. The integer is part of the numerical representation used to express a fixed-point number. Fixed-point numbers can be represented as

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

where the slope can be expressed as

$$\textit{slope} = \textit{fractional slope} \times 2^{\textit{exponent}}$$

See Also bias, fixed-point representation, fractional slope, integer, slope

integer length

The number of bits to the left of the binary point in a fixed-point representation of a number.

See Also binary point, bit, fixed-point representation, fraction length, integer

least significant bit (LSB)

The bit in a binary word that can represent the smallest value. The LSB is the rightmost bit in a big-endian-ordered binary word.

See Also big-endian, binary word, bit, most significant bit

logging

A tool provided by the **Fixed-Point Settings** interface that outputs the minimum values, maximum values, and any overflows for all Fixed-Point Blockset blocks in any model that you run with a fixed-point license.

See Also overflow

logical shift

A shift of the bits of a binary word, for which a zero is incorporated into the most significant bit for each bit shift to the right and into the least significant bit for each bit shift to the left.

See Also arithmetic shift, binary point, binary word, bit, most significant bit

mantissa

Part of the numerical representation used to express a floating-point number. Floating-point numbers are typically represented as

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

See Also exponent, floating-point representation

most significant bit (MSB)

The bit in a binary word that can represent the largest value. The MSB is the leftmost bit in a big-endian-ordered binary word.

See Also big-endian, binary word, bit, least significant bit

nearest (round toward)

A rounding mode that rounds to the closest representable number, with the exact midpoint rounded to the closest representable number in the direction of positive infinity.

See Also ceiling (round toward), floor (round toward), rounding, truncation, zero (round toward)

noncontiguous binary point

A binary point that is understood to fall outside of the word length of a data type. For example, the binary point for the following 4-bit word is understood to occur two bits to the right of the word length

0000_ _.

thereby giving the bits of the word the following potential values:

$2^5 2^4 2^3 2^2$_ _.

See Also binary point, data type, word length

one's complement representation

A representation of signed fixed-point numbers. Negating a binary number in one's complement requires a bitwise complement. That is, all 0's are flipped to 1's and all 1's are flipped to 0's. In one's complement notation there are two ways to represent zero. A binary word of all 0's represents "positive" zero, while a binary word of all 1's represents "negative" zero.

See Also binary number, binary word, sign/magnitude representation, signed fixed-point, two's complement representation

overflow

The situation that occurs when the magnitude of a calculation result is too large for the range of the data type being used. In many cases you can choose to either saturate or wrap overflows.

See Also saturation, wrapping

padding

Extending the least significant bit of a binary word with one or more zeros.

S*ee Also* least significant bit

precision

1. A measure of the smallest numerical interval that a fixed-point data type and scaling can represent, determined by the value of the number's least significant bit. The precision is given by the slope, or the number of fractional bits. The term "resolution" is sometimes used as a synonym for this definition.

2. A measure of the difference between a real-world numerical value and the value of its quantized representation. This is sometimes called quantization error or quantization noise.

See Also data type, fraction, least significant bit, quantization, quantization error, range, slope

Q format

A representation used by Texas Instruments to encode signed two's complement fixed-point data types. This fixed-point notation takes the form

Qm.n

where

- *Q* indicates that the number is in Q format.
- *m* is the number of bits used to designate the two's complement integer part of the number.
- *n* is the number of bits used to designate the two's complement fractional part of the number, or the number of bits to the right of the binary point.

In Q format notation, the most significant bit is assumed to be the sign bit.

See Also binary point, bit, data type, fixed-point representation, fraction, integer, two's complement

quantization

The representation of a value by a data type that has too few bits to represent it exactly.

See Also bit, data type, quantization error

quantization error

The error introduced when a value is represented by a data type that has too few bits to represent it exactly, or when a value is converted from one data type to a shorter data type. Quantization error is also called quantization noise.

See Also bit, data type, quantization

radix point

A symbol in the shape of a period that separates the integer and fractional parts of a number in any base system. Bits to the left of the radix point are integer and/or sign bits, and bits to the right of the radix point are fraction bits.

See Also binary point, bit, fraction, integer, sign bit

range

The span of numbers that a certain data type can represent.

See Also data type, precision

resolution

See **precision**

rounding

Limiting the number of bits required to express a number. One or more least significant bit(s) are dropped, resulting in a loss of precision. Rounding is necessary when a value can not be expressed exactly by the number of bits designated to represent it.

See Also bit, ceiling (round toward), floor (round toward), least significant bit, nearest (round toward), precision, truncation, zero (round toward)

saturation

A method of handling numeric overflow that represents positive overflows as the largest positive number in the range of the data type being used, and negative overflows as the largest negative number in the range.

See Also overflow, wrapping

scaling

1. The format used for a fixed-point number of a given word length and signedness. The slope and bias together form the scaling of a fixed-point number.

2. Changing the slope and/or bias of a fixed-point number without changing the stored integer.

See Also bias, fixed-point representation, integer, slope

shift

A movement of the bits of a binary word either towards the most significant bit ("to the left") or towards the least significant bit ("to the right"). Shifts to the right may either be logical, where the spaces emptied at the front of the word with each shift are filled in with zeros, or arithmetic, where the word is sign extended as it is shifted to the right.

See Also arithmetic shift, logical shift, sign extension

sign bit

The bit (or bits) in a signed binary number that indicates whether the number is positive or negative.

See Also binary number, bit

sign extension

Adding additional bits that have the value of the most significant bit to the high end of a two's complement number. Sign extension does not change the value of the binary number.

See Also binary number, guard bits, most significant bit, two's complement representation, word

sign/magnitude

A representation of signed fixed-point or floating-point numbers. In sign/magnitude representation, one bit of a binary word is always the dedicated sign bit, while the remaining bits of the word encode the magnitude of the number. Negation using sign/magnitude representation consists of flipping the sign bit from 0 (positive) to 1 (negative), or from 1 to 0.

See Also binary word, bit, fixed-point representation, floating-point representation, one's complement representation, sign bit, signed fixed-point, two's complement representation

signed fixed-point

A fixed-point number or data type that can represent both positive and negative numbers.

See Also data type, fixed-point representation, unsigned fixed-point

slope

Part of the numerical representation used to express a fixed-point number. Along with the bias, the slope forms the scaling of a fixed-point number. Fixed-point numbers can be represented as

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

where the slope can be expressed as

$$\mathit{slope} = \mathit{fractional\,slope} \times 2^{\mathit{exponent}}$$

See Also bias, fixed-point representation, fractional slope, integer, scaling, [Slope Bias]

[Slope Bias]

A representation used by the Fixed-Point Blockset to define the scaling of a fixed-point number.

See Also bias, scaling, slope

truncation

A rounding mode that drops one or more least significant bits from a number.

See Also ceiling (round toward), floor (round toward), nearest (round toward), rounding, zero (round toward)

two's complement representation

A common representation of signed fixed-point numbers. Negation using signed two's complement representation consists of a translation into one's complement followed by the binary addition of a one.

See Also binary word, one's complement representation, sign/magnitude representation, signed fixed-point

unsigned fixed-point

A fixed-point number or data type that can only represent numbers greater than or equal to zero.

See Also data type, fixed-point representation, signed fixed-point

word

A fixed-length sequence of binary digits (1's and 0's). In digital hardware, numbers are stored in words. The way hardware components or software functions interpret this sequence of 1's and 0's is described by a data type.

See Also binary word, data type

word length

The number of bits in a binary word or data type.

See Also binary word, bit, data type

wrapping

A method of handling overflow. Wrapping uses modulo arithmetic to cast a number that falls outside of the representable range of the data type being used back into the representable range.

See Also data type, overflow, range, saturation

zero (round toward)

A rounding mode that rounds to the closest representable number in the direction of zero.

See Also ceiling (round toward), floor (round toward), nearest (round toward), rounding, truncation

Index