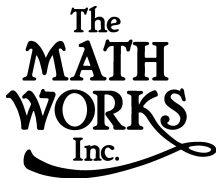
DSP Blockset

For Use with SIMULINK®

Modeling

Simulation

Implementation



User's Guide

Version 4

How to Contact The MathWorks:

508-647-7000 Phone

508-647-7001 Fax

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

`http://www.mathworks.com` Web
`ftp.mathworks.com` Anonymous FTP server
`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
`subscribe@mathworks.com` Subscribing user registration
`service@mathworks.com` Order status, license renewals, passcodes
`info@mathworks.com` Sales, pricing, and general information

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 (R10)
	January 1999	Fourth printing	DSP Blockset 3.0 (R11)
	November 2000	Fifth printing	DSP Blockset 4.0 (Release 12)

Contents

1 Introduction

2 Simulink and the DSP Blockset

3 Working with Signals

4 DSP Operations

5 DSP Block Reference

6 DSP Function Reference

1

Introduction

Welcome to the DSP Blockset

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

- "What Is the DSP Blockset?"
- "What Is in the DSP Blockset?"
- "Getting Started with the DSP Blockset"
- "R12 Related Products"

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

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

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

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

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

What Is the DSP Blockset?

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

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

Key Features

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

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

- "Frame-Based Operations"
- "Matrix Support"
- "Adaptive and Multirate Filtering"
- "Statistical Operations"
- "Linear Algebra"
- "Parametric Estimation"
- "Real-Time Code Generation"

Frame-Based Operations

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

The DSP Blockset delivers this same high level of performance for both simulation and code generation by incorporating frame-processing capability

into all of its blocks. A completely frame-based model can run several times faster than the same model processing sample-by-sample; faster still if data sources are frame based.

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

Matrix Support

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

- *General two-dimensional array*

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

- *Factored submatrices*

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

- *Multichannel frame-based signal*

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

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

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

Adaptive and Multirate Filtering

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

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

Statistical Operations

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

Linear Algebra

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

Parametric Estimation

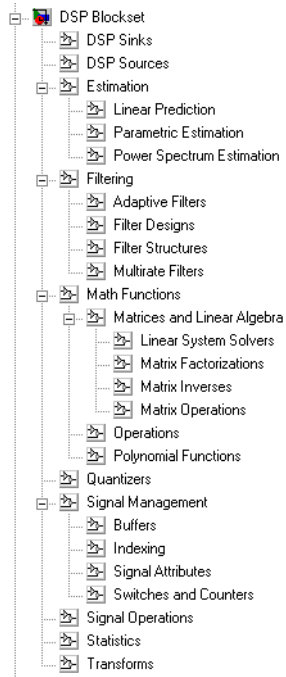
The Parametric Estimation library provides a number of methods for modeling a signal as the output of an AR system. The methods include the Burg AR Estimator, Covariance AR Estimator, Modified Covariance AR Estimator, and Yule-Walker AR Estimator, which allow you to compute the AR system parameters based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

Real-Time Code Generation

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

What Is in the DSP Blockset?

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



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

```
dsplib
```

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

Demos

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

For a complete list of all the blocks in the DSP Blockset by library, see "Block Library Contents" on page 5-5.

Installation

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

Getting Started with the DSP Blockset

To get started with the DSP Blockset, open the Simulink Library Browser by pressing the  button on the MATLAB toolbar, or by typing

```
simulink
```

at the command line. Expand the DSP Blockset library tree in the Block Browser by clicking the ⊞ symbol next to the **DSP Blockset** entry. You can drag blocks directly from the Library Browser into a Simulink model.

Alternatively, you can open the DSP Blockset in its own window by typing

```
dsplib
```

at the MATLAB command line. Double-click on any library in the DSP Blockset window to view its contents, and double-click on a block to access its parameter dialog box.

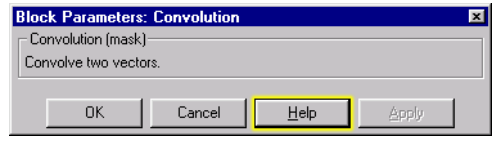
The following sections provide additional information to help get you started with the DSP Blockset:

- "How to Get Help Online"
- "How to Use This Guide"
- "Technical Conventions"
- "Typographical Conventions"

How to Get Help Online

There are a number of easy ways to get help on the DSP Blockset while you're working at the computer:

- *Block Help* – Press the **Help** button in any block dialog box to view the online reference documentation for that block.



- *Simulink Library Browser* – Right-click on a block to access the help for that block.

- *Help Browser* – Select **Help Browser** from the MATLAB **Help** or **View** menu (or type `doc` or `helpdesk` at the command line) to display the Help Browser. Select **DSP Blockset** in the **Contents** pane.
- *Command Line* – Type `doc('block name')` at the command line to access the help for a block with the name *block name*. Spaces and capitalization in the block name are ignored.
- *Help Desk* (*remote*) – Use a Web browser or the Help Browser to connect to the MathWorks Web site at `www.mathworks.com`. Follow the **Documentation** link on the **Support** Web page for remote access to the documentation.
- *Release Information* – Select **Release 12 New Features** in the **Contents** pane of the Help Browser to view information related to the version of the DSP Blockset that you are using, and to find out about recent changes to the blockset. You can also type `info dspblks` at the MATLAB command line to view detailed release information related to bug fixes and enhancements.

How to Use This Guide

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

- Read Chapter 2, “Simulink and the DSP Blockset,” to get an overview of fundamental Simulink and DSP Blockset concepts. Also see the Simulink documentation for more information on the Simulink environment.
- Read Chapter 3, “Working with Signals,” for details on key operations common to many signal processing tasks.
- Read Chapter 4, “DSP Operations,” for a discussion of important block applications.
- Read Chapter 5, “DSP Block Reference,” for a description of each block’s operation, parameters, and characteristics.
- Read the “DSP Blockset” sections of *R12 Release Notes* and to learn about enhancements made to the blockset in the current version.

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

Technical Conventions

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

- "Signal Dimension Nomenclature"
- "Frame-Based Signal Nomenclature"
- "Sampling Nomenclature"

Signal Dimension Nomenclature

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

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

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

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

This matrix can be represented in MATLAB notation as

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

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

Vectors. Strictly speaking, a Simulink *vector* is a one-dimensional (1-D) array of values, an ordered list that has no row or column orientation. For convenience, the DSP Blockset help uses the plain term *vector* to refer to any of the following three entities:

- One-dimensional array, also called a *1-D vector*
- 1-by-N matrix, also called a *row vector*
- M-by-1 matrix, also called a *column vector*

The *size* or *length* of a vector, M for a column vector or N for a row vector, is the number of elements that it contains. There is no MATLAB equivalent for a 1-D Simulink vector (i.e., all MATLAB vectors have either a row or column orientation), and most blocks in the DSP Blockset treat a 1-D vector as a column vector.

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

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

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

Frame-Based Signal Nomenclature

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

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

Sampling Nomenclature

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

Typographical Conventions

This manual uses some or all of these conventions.

Item	Convention to Use	Example
Example code	`Monospace font`	To assign the value 5 to `A`, enter `A = 5`
Function names/syntax	`Monospace font`	The `cos` function finds the cosine of each array element. Syntax line example is `MLGetVar ML_var_name`
Keys	**Boldface** with an initial capital letter	Press the **Return** 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 names, menu items, and controls	**Boldface** with an initial capital letter	Choose the **File** menu.
New terms	*Italics*	An *array* is an ordered collection of information.
String variables (from a finite list)	*`Monospace italics`*	`sysc = d2c(sysd,'method')`

R12 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 MATLAB’s capabilities. The blocksets all include blocks that extend Simulink’s capabilities. The DSP Blockset requires MATLAB 6.0, Simulink 4.0, and Signal Processing Toolbox 5.0.

Product	Description
Real-Time Workshop	Tool that generates customizable C code from Simulink models and automatically builds programs that can run in real time in a variety of environments
Signal Processing Toolbox	Tool for algorithm development, signal and linear system analysis, and time-series data modeling
Simulink	Interactive, graphical environment for modeling, simulating, and prototyping dynamic systems

Product	Description
xPC Target	Tool for adding I/O blocks to Simulink block diagrams and downloading the code generated by Real-Time Workshop to a second PC that runs the xPC Target real-time kernel, for rapid prototyping and hardware-in-the-loop testing of control and DSP systems
Motorola DSP Developer's Kit	Tool for co-development and co-simulation of Motorola 56300 and 56600 fixed-point assembly code

2

Simulink and the DSP Blockset

Overview

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

- "The Simulink Environment"
- "Configuring Simulink for DSP Systems"

The first section provides a brief overview of the Simulink environment. The second section provides guidance in tailoring the environment for DSP system simulation.

The Simulink Environment

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

This section includes the following topics:

- "Starting Simulink"
- "Getting Started with Simulink"
- "Learning More About Simulink"

Starting Simulink

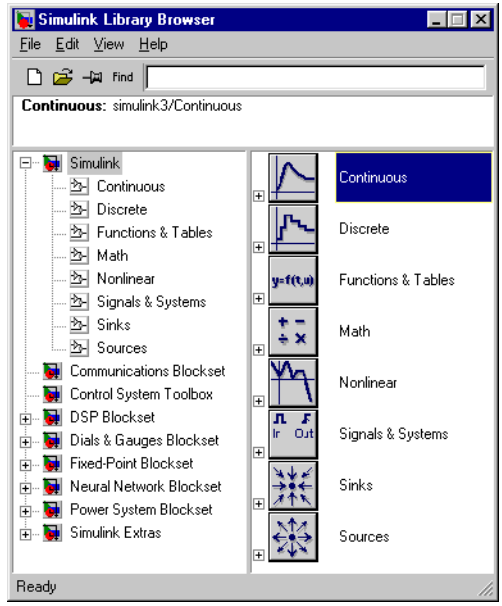
To start Simulink, click the  icon in the MATLAB toolbar, or type

```
simulink
```

at the command line.

Simulink on PC Platforms

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

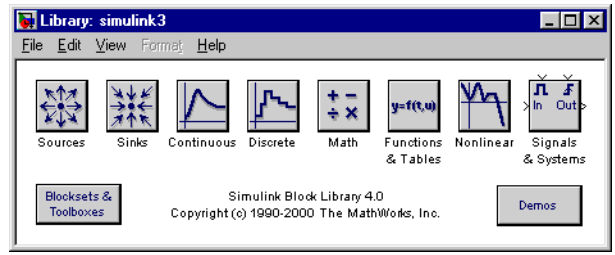


The first item in the list is the Simulink blockset itself, which is already expanded to show the available Simulink libraries. Click the ⊞ symbol to the left of any blockset name to expand the hierarchical list and display that blockset's libraries within the browser.

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

Simulink on UNIX Platforms

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



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

The Simulink Libraries

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

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

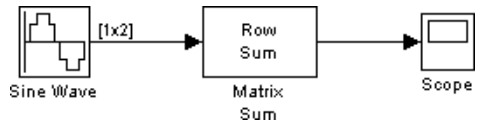
Getting Started with Simulink

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

- “Model Definition”
- “Model Simulation”

Model Definition

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



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

   One of the things that `dspstartup` does is set the **Stop time** value in the **Simulation parameters** dialog box to `inf` for all new models. The `inf` setting instructs Simulink to run the model for as long as the computer's memory allows. You can access this dialog box and enter a different **Stop time** value by selecting **Simulation parameters** from the model window's **Simulation** menu.

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

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

4. Add a Sine Wave block to the model.

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

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

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

5. Add a Matrix Sum block to the model.

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

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

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

   d. Drag the Matrix Sum block into the model window.

6 Add a Scope block to the model.

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

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

7 Connect the blocks.

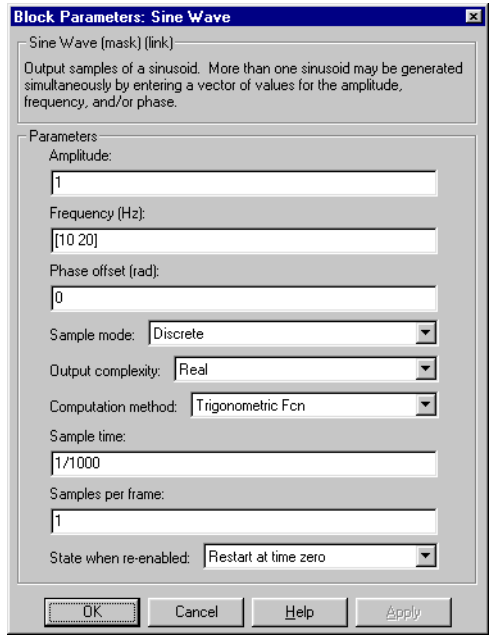
   a Position the pointer near the output port of the Sine Wave block. Hold down the mouse button (the left button for a multibutton mouse) and drag the line that appears until it touches the input port of the Matrix Sum block. Release the mouse button.

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

8 Set the block parameters.

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

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



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

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

You can now move on to the model simulation phase.

Model Simulation

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

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

At each sample time, the output matrix contains one sample from each of the two sinusoids. The Matrix Sum block adds the two matrix elements together

to produce a scalar output. Thus, the input to the Scope block is the point-by-point sum of the two sinusoids.

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

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

4. Adjust the Scope block's display.

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

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

5. Vary the Sine Wave block parameters.

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

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

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

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

Many parameters *cannot* be changed while a simulation is running. This is usually the case for parameters that directly or indirectly alter a signal's dimensions or sample rate. There are some parameters, however, like the Sine Wave **Frequency** parameter, that you can *tune* without terminating the simulation. In the online "DSP Block Reference" these parameters are marked "Tunable," indicating that they are *tunable* while the simulation runs.

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

Learning More About Simulink

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

- Browse through the Simulink documentation to get complete exposure to all of Simulink's capabilities.
- Open the Simulink library as described in "Starting Simulink" on page 2-3. Build a few simple models using blocks from the Simulink and DSP Blockset libraries.
- Open some of the models in the DSP Blockset Demos library. Most of the advanced demos have blocks that you can double-click to get information about the algorithm or implementation. The Demos library also contains easy-to-understand models that demonstrate some of the blockset's elementary math and statistics blocks. In each case, just select **Start** from the **Simulation** menu to run the simulation.

Configuring Simulink for DSP Systems

When you create a new DSP model, you may want to adjust certain Simulink settings to suit your own needs. A typical change, for example, is to adjust the **Stop time** parameter (in the **Simulation Parameters** dialog box) to a different value. Another common change is to specify the **Fixed-step** option in the **Solver options** panel to reflect the discrete-time nature of the DSP model.

The DSP Blockset provides an M-file, `dspstartup`, that lets you automate this configuration process so that every new model you create is preconfigured for DSP simulation. The M-file executes the following commands.

```
set_param(0, ...
   'SingleTaskRateTransMsg','error', ...
   'Solver',                'fixedstepdiscrete', ...
   'SolverMode',            'SingleTasking', ...
   'StartTime',             '0.0', ...
   'StopTime',              'inf', ...
   'FixedStep',             'auto', ...
   'SaveTime',              'off', ...
   'SaveOutput',            'off', ...
   'AlgebraicLoopMsg',      'error', ...
   'InvariantConstants',    'on', ...
   'RTWOptions',         [get_param(0,'RTWOptions')
                           ' -aRollThreshold=2']);
```

The following sections provide information about `dspstartup`:

- "Using dspstartup.m"
- "Customizing dspstartup.m"
- "Performance-Related Settings"
- "Miscellaneous Settings"

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 launch 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

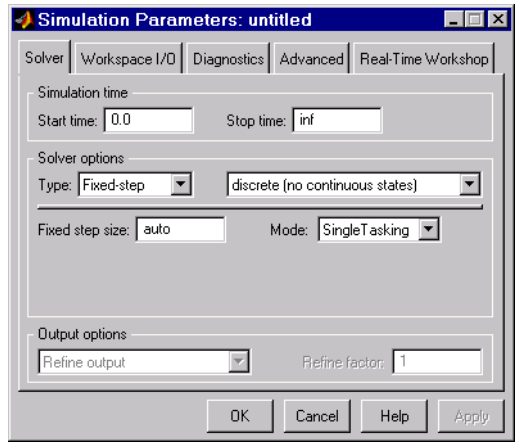
You can edit the dspstartup M-file to change any of the settings above or to add your own custom settings. For example, you can change the 'StopTime' option to a value that is better suited to your particular simulations, or set the 'SaveTime' option to 'on' if you prefer to record the simulation sample times.

Performance-Related Settings

A number of the settings in the `dspstartup` M-file are chosen to improve the performance of the simulation:

- `'SaveTime'` is set to `'off'`

  When `'SaveTime'` is set to `'off'`, Simulink does not save the `tout` time-step vector to the workspace. The time-step record is not usually needed for analyzing discrete-time simulations, and disabling it saves a considerable amount of memory, especially when the simulation runs for an extended period of time. To enable time recording for a particular model, select the **Time** check box in the **Workspace I/O** panel of the **Simulation Parameters** dialog box (shown below).



- `'SaveOutput'` is set to `'off'`

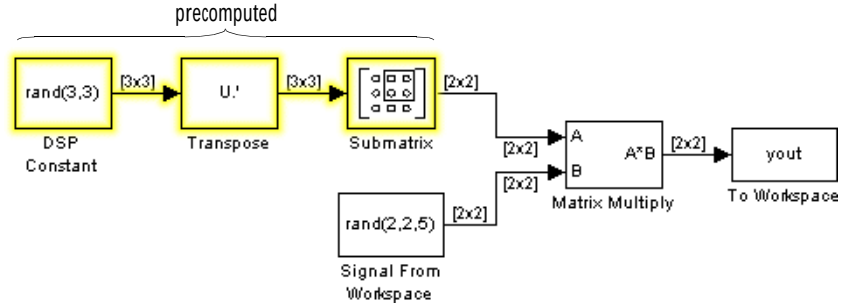
  When `'SaveOutput'` is set to `'off'`, Simulink Outport blocks in the top level of a model do not generate an output (`yout`) in the workspace. To reenable output recording for a particular model, select the **Output** check box in the **Workspace I/O** panel of the **Simulation Parameters** dialog box (above).

- `'InvariantConstants'` is set to `'on'`

  When `'InvariantConstants'` is set to `'on'`, Simulink precomputes the values of all constant blocks (e.g., DSP Constant, Constant Diagonal Matrix) at the start of the simulation, and does not update them again for the

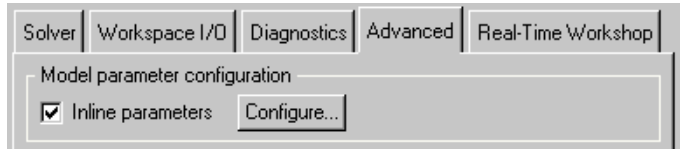
duration of the simulation. Simulink additionally precomputes the outputs of all downstream blocks driven exclusively by constant blocks.

In the example below, the input to the top port (U) of the Matrix Multiply block is computed only once, at the start of the simulation.



This eliminates the computational overhead of continuously reevaluating these constant branches, which in turn results in faster simulation, and smaller and more efficient generated code.

Note, however, that when `'InvariantConstants'` is set to `'on'`, changes that you make to parameters in a constant block while the simulation is running are not registered by Simulink, and do not affect the simulation. If you would like to adjust the model constants while the simulation is running, you can turn off `'InvariantConstants'` by deselecting the **Inline Parameters** check box in the **Advanced** panel of the **Simulation Parameters** dialog box.



- `'RTWOptions'` sets loop-rolling threshold to 2

  By default, the Real-Time Workshop "unrolls" a given loop into inline code when the number of loop iterations is less than five. This avoids the overhead

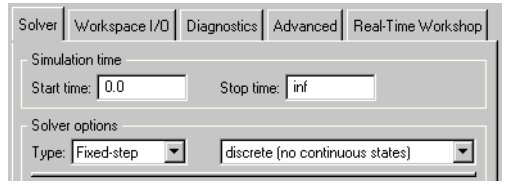
of servicing the loop in cases when inline code can be used with only a modest increase in the file size.

However, because typical DSP processors offer *zero-overhead looping*, code size is the primary optimization constraint in most designs. It is therefore more efficient to minimize code size by generating a loop for every instance of iteration, regardless of the number of repetitions. This is what the `'RTWOptions'` loop-rolling setting in `dspstartup` accomplishes.

Miscellaneous Settings

The `dspstartup` M-file adjusts several other parameters to make it easier to run DSP simulations. Two of the important settings are:

- `'StopTime'` is set to `'inf'`, which allows the simulation to run until you manually stop it by selecting **Stop** from the **Simulation** menu, or by pressing the **Stop Simulation** button on the toolbar. To set a finite stop time, enter a value for the **Stop time** parameter in the **Simulation Parameters** dialog box.



- `'Solver'` is set to `'fixedstepdiscrete'`, which selects the fixed-step solver option instead of Simulink's default variable-step solver. See "Discrete-Time Signals" on page 3-3 for more information about the various solver settings.

3

Working with Signals

Overview

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

- "Signal Concepts"
- "Sample Rates and Frame Rates"

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

- "Creating Signals"
- "Constructing Signals"
- "Deconstructing Signals"
- "Importing Signals"
- "Exporting Signals"
- "Viewing Signals"

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

- "Delay and Latency"

Signal Concepts

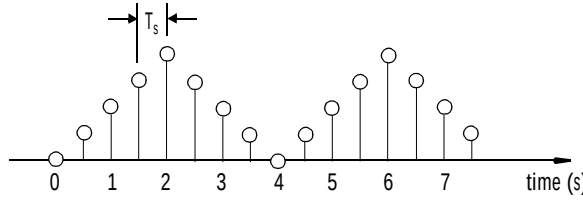
Simulink models can process both discrete-time and continuous-time signals, although models that are built with the DSP Blockset are often intended to process only discrete-time signals. The next few sections cover the following topics:

- "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" – An explanation of how continuous-time signals are treated by various blocks in the DSP Blockset
- "Multichannel Signals" – A description of how multichannel signals are represented in Simulink
- "Benefits of Frame-Based Processing" – 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 called the signal's *sample times*; traditionally, a discrete-time signal is considered to be undefined at points in time between these instants. For a periodically sampled signal, the equal interval between any pair of sample times is the signal's *sample period*, T_s. The *sample rate*, F_s, is the reciprocal of the sample period, or $1/T_s$.

For example, the 7.5-second triangle wave segment below has a sample period of 0.5 seconds, 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"

Time and Frequency Terminology

A number of different terms are used to describe the characteristics of discrete-time signals found in Simulink models. These terms, which are listed in the table below, are frequently used in Chapter 5, "DSP Block Reference," to describe the way that various blocks operate on sample-based and frame-based signals.

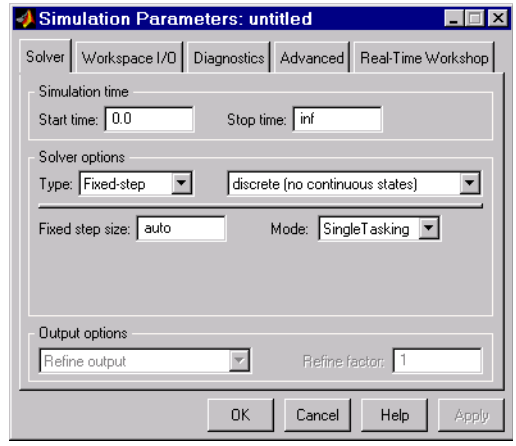
Term	Symbol	Units	Notes
Sample period	T_s T_{si} T_{so},	Seconds	The time interval between consecutive samples in a sequence, as the input to a block (T_{si}) or the output from a block (T_{so}).
Frame period	T_f T_{fi} T_{fo}	Seconds	The time interval between consecutive frames in a sequence, as the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

Term	Symbol	Units	Notes
Normalized frequency	f_n	Two cycles per sample	Frequency (linear) of a periodic signal normalized to half the sample rate, $f_n = \omega/\pi = 2f/F_s$.
Angular frequency	Ω	Radians per second	Frequency of a periodic signal in angular units, $\Omega = 2\pi f$.
Digital (normalized angular) frequency	ω	Radians per sample	Frequency (angular) of a periodic signal normalized to the sample rate, $\omega = \Omega/F_s = \pi f_n$.

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

Discrete-Time Signals in Simulink

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



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

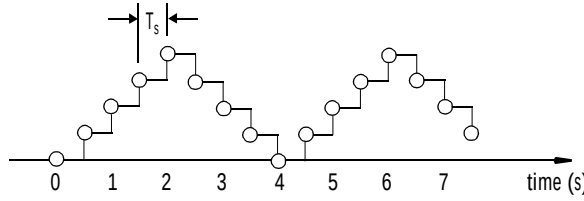
- "Recommended Settings for Discrete-Time Simulations"
- "Sample Time Offsets"
- "Cross-Rate Operations in Variable-Step and Fixed-Step SingleTasking Modes"
- "Sample Time Offsets"

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

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

You can automatically set the above solver options for all new models by running the `dspstartup` M-file. See "Configuring Simulink for DSP Systems" on page 2-11 for more information.

In **Fixed-step SingleTasking** mode, discrete-time signals *differ* from the prototype described in "Discrete-Time Signals" on page 3-3 by remaining *defined* between sample times. For example, the representation of the discrete-time triangle wave looks like this.



The above signal's value at t=3.112 seconds is the same as the signal's value at t=3 seconds. In this 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 is frozen at its last value.

As a result, in **Fixed-step SingleTasking** mode, Simulink permits cross-rate operations such as the addition of two signals of different rates. This is explained further in "Cross-Rate Operations in Variable-Step and Fixed-Step SingleTasking Modes" on page 3-7.

Additional Settings for Discrete-Time Simulations. It is worthwhile to know how the other solver options available in Simulink affect discrete-time signals. In particular, you should be aware of the properties of discrete-time signals under the following settings:

- **Type** = **Fixed-step**, **Mode** = **MultiTasking**
- **Type** = **Variable-step** (Simulink's default solver)
- **Type** = **Fixed-step**, **Mode** = **Auto**

When the fixed-step multi-tasking solver is selected, discrete signals in Simulink most accurately model the prototypical discrete signal described in "Discrete-Time Signals" on page 3-3. In particular, when these settings are in effect, discrete signals are *undefined* between sample times. Simulink generates an error when operations attempt to reference the undefined region of a signal, as, for example, when signals with different sample rates are added.

To perform cross-rate operations like the addition of two signals with different sample rates, you must *explicitly* convert the two signals to a common sample rate. There are several blocks provided for precisely this purpose in the Signal Operations and Multirate Filters libraries. See "Converting Sample Rates and Frame Rates" on page 3-20 for more information. By requiring explicit rate conversions for cross-rate operations in discrete mode, Simulink helps you to identify sample rate conversion issues early in the design process.

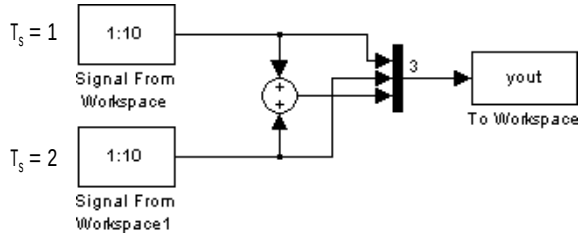
When the variable-step solver is selected, discrete time signals remain defined between sample times, just as in the **Fixed-step SingleTasking** setting described in "Recommended Settings for Discrete-Time Simulations" above. Thus, in this mode, cross-rate operations are allowed by Simulink.

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

Cross-Rate Operations in Variable-Step and Fixed-Step SingleTasking Modes. In Simulink's **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 distinct from the signal's own rate and phase, you will still measure meaningful values. Note that in the recommended `dspstartup` settings, cross-rate operations generate an error even though the

solver is in fixed-step single-tasking mode. This results from the **Error** setting for **SingleTask rate transition** under the Diagnostics pane in the Simulation Parameters dialog box.

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, ...



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
```

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. Note that this example will generate an error under the dspstartup settings due to the **Error** setting for **SingleTask rate transition** in the Diagnostics pane in the **Simulation Parameters** dialog box.

In general, for **Variable-step** and **Fixed-step SingleTasking** modes, when you measure the value of a discrete signal in-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"

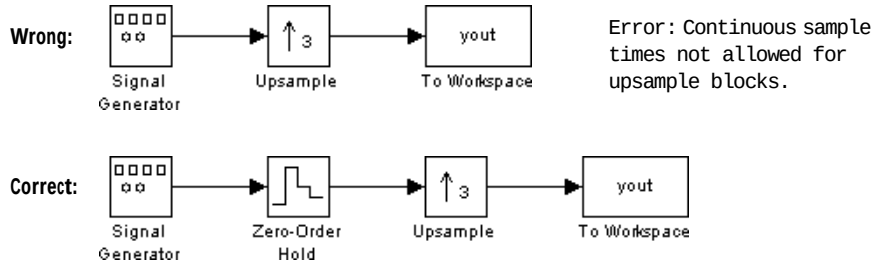
See Chapter 5, "DSP Block Reference," for information about the particular sample characteristics of each block in the blockset.

Source Blocks

Source blocks are those blocks that generate or import signals in a model. Many of these blocks have the term "from workspace" or "constant" in the block name (e.g., Signal From Workspace, DSP Constant), and most appear in the DSP Sources library. See section "Importing Signals" on page 3-62 to fully explore the features of these blocks.

Continuous-Time Source Blocks. The sample period for continuous-time source blocks is set internally to zero (which indicates a continuous-time signal). An example is Simulink's Signal Generator block. Continuous-time signals are rendered in black when **Sample time colors** is selected from the **Format** menu. As shown below, when connecting such blocks to certain nonsource discrete-time blocks, you may need to interpose a Zero-Order Hold block to

discretize the signal. Specify the desired sample period for the signal in the **Sample time** parameter of the Zero-Order Hold block.



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

Discrete-Time Source Blocks. Discrete-time source blocks, such as Signal From Workspace, require a discrete (i.e., 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

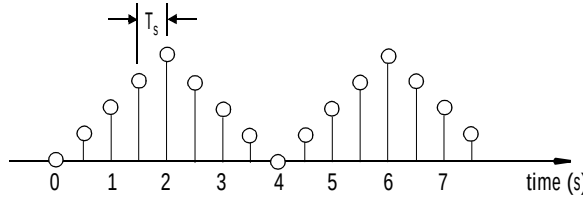
All nonsource blocks in the DSP Blockset accept discrete signals, and inherit the sample period of the input. Others additionally accept continuous-time discrete signals.

Discrete-Time Nonsource Blocks. Many blocks can accept only discrete-time inputs, and generate only discrete-time outputs. Examples are all of the resampling and delay blocks (e.g., Upsample, Integer Delay), which *inherit* the sample period of the driving block (the block supplying the input). This means that the block automatically synchronizes its sampling rate with the driving block. For example, if the driving block's sample period is 0.5 seconds, then 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 Complex Exponential and dB Gain. The nonsource *triggered* blocks (e.g., Triggered Delay Line) are also in this category.

Multichannel Signals

The figure below shows the prototypical discrete-time signal discussed in "Discrete-Time Signals" on page 3-3. If this signal were propagated through a model as shown, 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:

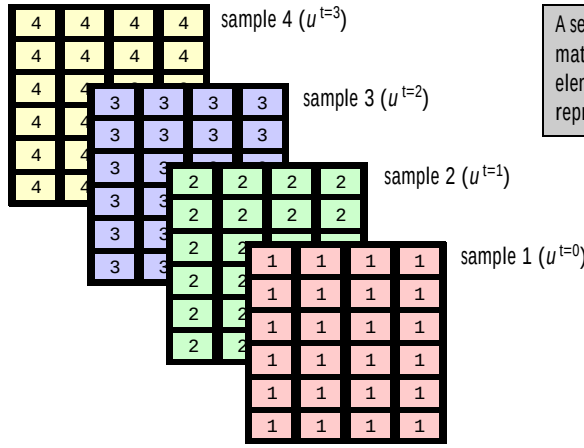
- "Sample-Based Multichannel Signals"
- "Frame-Based Multichannel Signals"

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

Sample-Based Multichannel Signals

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

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



Then the signal in channel 1 is composed of the following sequence.

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

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

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

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

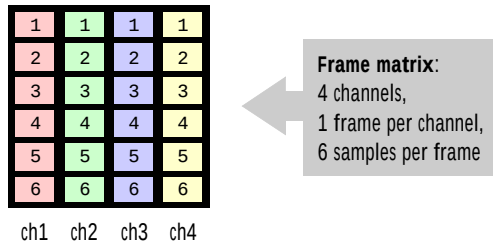
- "Creating Signals" on page 3-33
- "Constructing Signals" on page 3-42
- "Deconstructing Signals" on page 3-54
- "Importing Signals" on page 3-62
- "Exporting Signals" on page 3-72
- "Viewing Signals" on page 3-80

Frame-Based Multichannel Signals

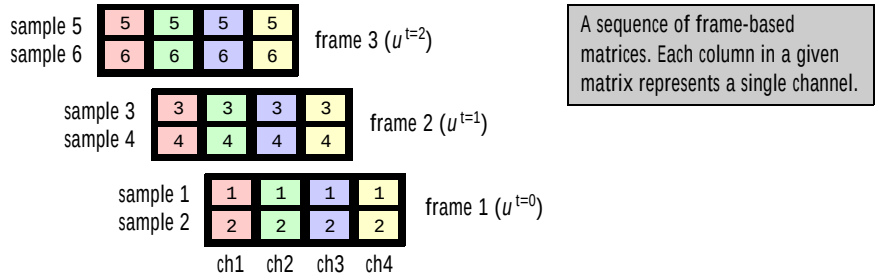
Frame-based multichannel signals are also represented as matrices. An M-by-N frame-based matrix represents M consecutive samples from each of N independent channels. In other words, each matrix *row* represents one sample

(or time slice) from N distinct signal channels, and each matrix *column* represents M consecutive samples from a single channel.

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



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



Then the signal in channel 1 is the following sequence.

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

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

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

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

- “Creating Signals” on page 3-33
- “Constructing Signals” on page 3-42
- “Deconstructing Signals” on page 3-54

- "Importing Signals" on page 3-62
- "Exporting Signals" on page 3-72
- "Viewing Signals" on page 3-80

Benefits of Frame-Based Processing

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

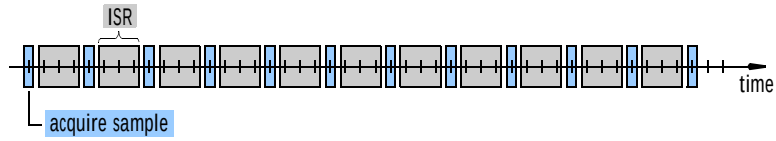
Accelerating Real-Time Systems

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

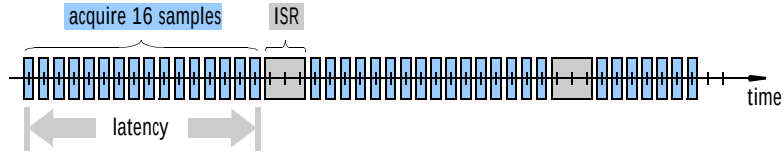
The figure below illustrates how throughput is increased by frame-based data acquisition. The thin blocks each represent the time elapsed during acquisition of a sample. The thicker blocks each represent the time elapsed during the interrupt service routine (ISR) that reads the data from the hardware.

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

Sample-based operation



Frame-based operation



It's important to note that frame-based processing may introduce a certain amount of latency into a process due to the inherent lag in buffering the initial frame. In most 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"
- "Converting Sample Rates and Frame Rates"
- "Changing Frame Status"

Sample Rate and Frame Rate Concepts

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

In contrast, the sample period, T_s, is the time interval between individual samples in a frame, which is necessarily shorter than the frame period when the frame size is greater than 1. The sample period of a frame-based signal is the quotient of the frame period and the frame size, M.

$$T_s = T_f / M$$

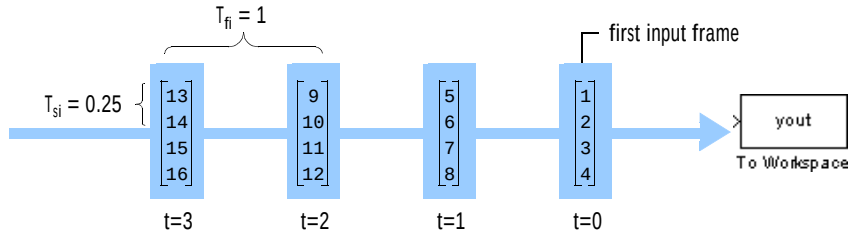
More specifically, the sample periods of inputs (T_{si}) and outputs (T_{so}) are related to their respective frame periods by

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

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

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

The illustration below shows a one-channel frame-based signal with a frame size (M_i) of 4 and a frame period (T_{fi}) of 1. The sample period, T_{si}, is therefore 1/4, or 0.25 seconds. A Probe block connected to this signal would display the frame period $T_{fi} = 1$.



In most cases, the sequence sample period T_{si} is of primary interest, while the frame rate is simply a consequence of the frame size that you choose for the signal. For a sequence with a given sample period, a larger frame size corresponds to a slower frame rate, and vice versa.

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

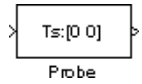
Inspecting Sample Rates and Frame Rates

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

- “Using the Probe Block to Inspect Rates”
- “Using Sample Time Color Coding to Inspect Sample Rates”

Using the Probe Block to Inspect Rates

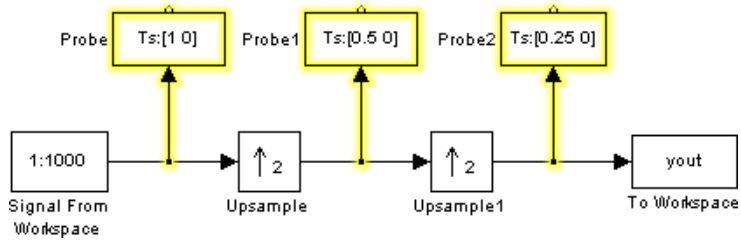
Connect Simulink’s Probe block to any line to display the period of the signal on that line. The period is displayed in the block icon itself (together with the line width and data type, if desired), making it easy to verify that the sample rates in the model are what you expect them to be. When the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



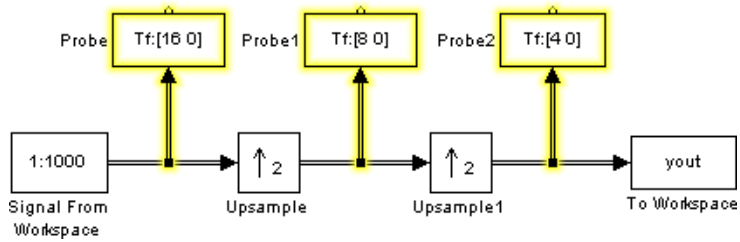
The block displays the label `Ts` or `Tf`, followed by a two-element vector. The first (left) element is the period of the signal being measured. The second (right) is the signal's sample time offset, which is usually `0`, as explained in "Sample Time Offsets" on page 3-9.

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

Probe Block Example: Sample-Based. The three Probe blocks in the sample-based model below verify that the signal's sample period is halved with each upsample operation: The output from the Signal From Workspace block has a sample period of 1 second, the output from the first Upsample block has a sample period of 0.5 seconds, and the output from the second Upsample block has a sample period of 0.25 seconds.



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

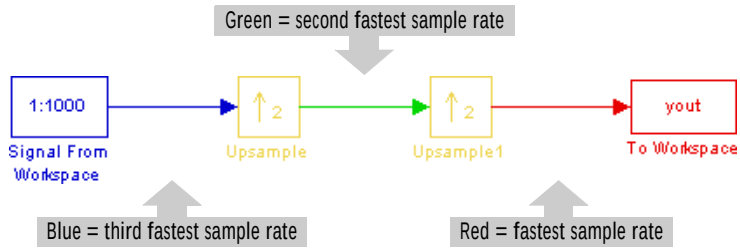


Note that the sample rate conversion is implemented through a change in the frame period rather than the frame size. This is because the **Frame-based mode** parameter in the Upsample blocks is set to **Maintain input frame size** rather than **Maintain input frame rate**. See "Converting Sample Rates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

Using Sample Time Color Coding to Inspect Sample Rates

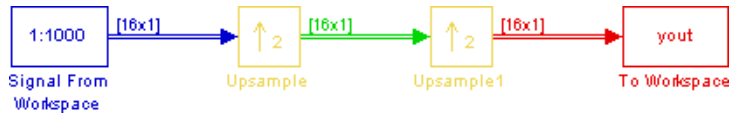
Turn on Simulink's sample time color coding option by selecting **Sample time colors** from the **Format** menu. For sample-based signals, this assigns each sample rate a different color. For frame-based signals, this assigns each frame rate a different color.

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



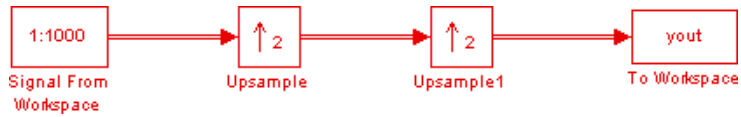
Since every sample-based signal in this model has a different sample rate, each signal is assigned a different color.

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



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

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



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

Converting Sample Rates and Frame Rates

In a DSP Blockset model, there are two types of periods that you will commonly be concerned with: sample periods and frame periods. The input and output sample periods of a block (T_{si} and T_{so}, respectively) are related to the input and output frame periods (T_{fi} and T_{fo}, respectively) by

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

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

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

The buffering and rate-conversion capabilities of the DSP Blockset generally allow you to independently vary any two of the three parameters (T_{so}, T_{fo}, M_o). In most cases, the sample period and the frame size are the two parameters of primary interest; the frame period is simply a consequence of your choices for the other two.

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

- Direct rate conversions

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

- Frame rebuffering

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

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

- "Direct Rate Conversion"
- "Frame Rebuffering"
- "Avoiding Unintended Rate Conversions"

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

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

Direct Rate Conversion

Rate conversion blocks accept an input signal at one sample rate, and propagate the same signal at a new sample rate. Several of these blocks contain a **Frame-based mode** parameter offering two options for adjusting the sample rate of the signal:

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

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

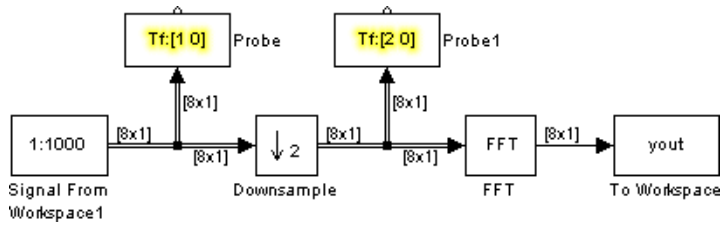
Rate Conversion Blocks. The following table lists the principal rate conversion blocks in the DSP Blockset. Blocks marked with an asterisk (*) offer the option of changing the rate by either adjusting the frame size or frame rate.

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

The following examples illustrate the two sample rate conversion modes:

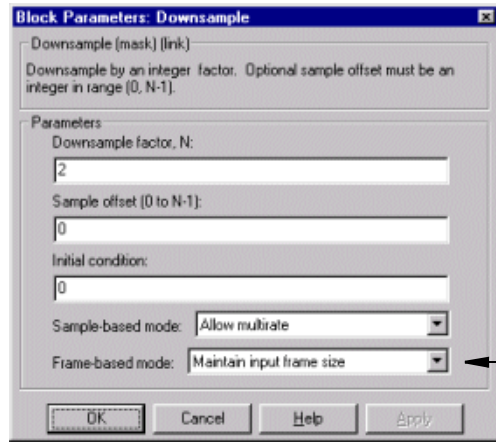
- "Example: Rate Conversion by Frame-Rate Adjustment"
- "Example: Rate Conversion by Frame-Size Adjustment"

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 seconds and 8 samples per frame, respectively, by the **Sample time** and **Samples per frame** parameters in the Signal From Workspace block. This results in a frame rate of 1 second (0.125∗8), as shown by the first Probe block.

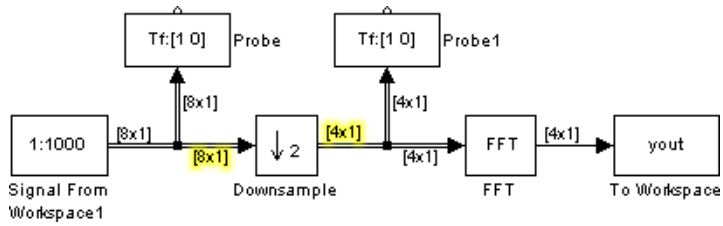
The Downsample block is configured to downsample the signal by changing the frame *rate* rather than the frame *size*. The dialog box with this setting is shown below.



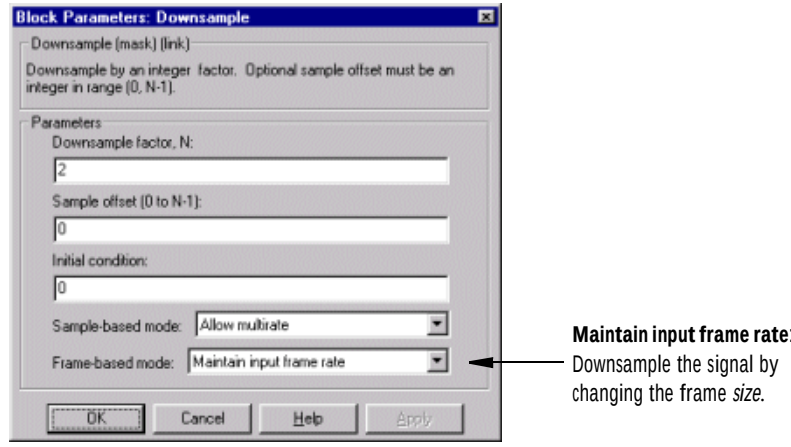
Maintain input frame size: Downsample the signal by changing the frame *rate*.

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

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



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



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 seconds 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"

Buffering with Preservation of the Signal. There are various reasons that you may need to rebuffer a signal to a new frame size at some point in a model. For example, your data acquisition hardware may internally buffer the sampled signal to a frame size that is not optimal for the DSP algorithm in the model. In this case, you would want to rebuffer the signal to a frame size more appropriate for the intended operations, but without introducing any change to the data or sample rate.

There are two blocks in the Buffers library that can be used to change a signal's frame size without altering the signal itself:

- Buffer – redistributes signal samples to a larger or smaller frame size
- Unbuffer – unbuffers a frame-based signal to a sample-based signal (frame size = 1)

The Buffer block preserves the signal's data and sample period only when its **Buffer overlap** parameter is set to `0`. The output frame period, T_{fo}, is

$$T_{fo} = \frac{M_o T_{fi}}{M_i}$$

where T_{fi} is the input frame period, M_i is the input frame size, and M_o is the output frame size specified by the **Buffer size** parameter.

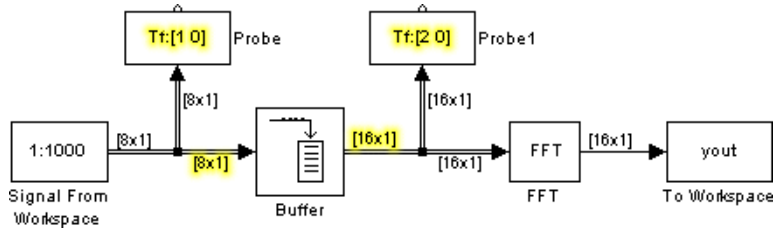
The Unbuffer block is specialized for completely unbuffering a frame-based signal to its sample-based equivalent, and always preserves the signal's data and sample period:

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

where T_{fi} and M_i are the period and size, respectively, of the frame-based input.

Both the Buffer and Unbuffer blocks preserve the sample period of the sequence in the conversion ($T_{so} = T_{si}$).

Example: Buffering with Preservation of the Signal. In the model below, a signal with a sample period of 0.125 seconds is rebuffered from a frame size of 8 to a frame size of 16. This doubles the frame period from 1 to 2 seconds, but does not change the sample period of the signal ($T_{so} = T_{si} = 0.125$).



Buffering with Alteration of the Signal. Some forms of buffering alter the signal's data or sample period, in addition to adjusting the frame size. There are many instances when this type of buffering is desirable, for example when creating sliding windows by overlapping consecutive frames of a signal, or selecting a subset of samples from each input frame for processing.

The blocks that alter a signal while adjusting its frame size are listed below. In this list, T_{si} is the input sequence sample period, and T_{fi} and T_{fo} are the input and output frame periods, respectively.

- Buffer adds duplicate samples to a sequence when the **Buffer overlap** parameter, L, is set to a nonzero value. The output frame period is related to the input sample period by

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

  where M_o is the output frame size specified by the **Buffer size** parameter. As a result, the new output sample period is

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

- Delay Line adds duplicate samples to the sequence when the **Delay line size** parameter, M_o, is greater than 1. The output and input frame periods are the same, $T_{fo} = T_{fi} = T_{si}$, and the new output sample period is

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

- Variable Selector can remove, add, and/or rearrange samples in the input frame when **Select** is set to **Rows**. The output and input frame periods are the same, $T_{fo} = T_{fi}$, and the new output sample period is

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

  where M_o is the length of the block's output, determined by the **Elements** vector.

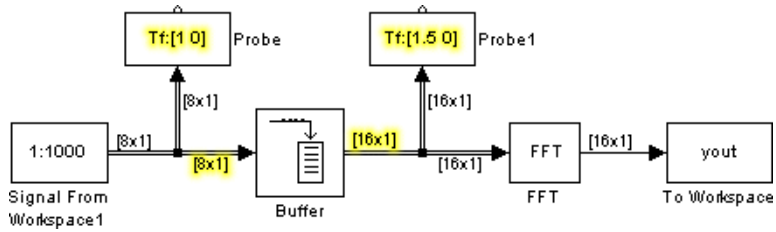
- Zero Pad adds samples to the sequence by appending zeros to each frame when **Zero pad along** is set to **Columns**. The output and input frame periods are the same, $T_{fo} = T_{fi}$, and the new output sample period is

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

  where M_o is the length of the block's output, determined by the **Number of output rows** parameter.

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

Example: Buffering with Alteration of the Signal. In the model below, a signal with a sample period of 0.125 seconds 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

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

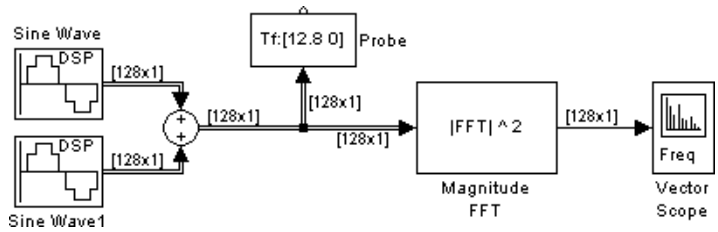
which indicates that T_{fo} should be (16-4)*0.125, or 1.5 seconds, as confirmed by the second Probe block. The sample period of the signal at the output of the Buffer block is no longer 0.125 seconds, but rather 0.0938 seconds (i.e., 1.5/16). Thus, both the signal's data and the signal's sample period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

Avoiding Unintended Rate Conversions

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

- "Example 1: No Rate Conversion"
- "Example 2: Unintended Rate Conversion"

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



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

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

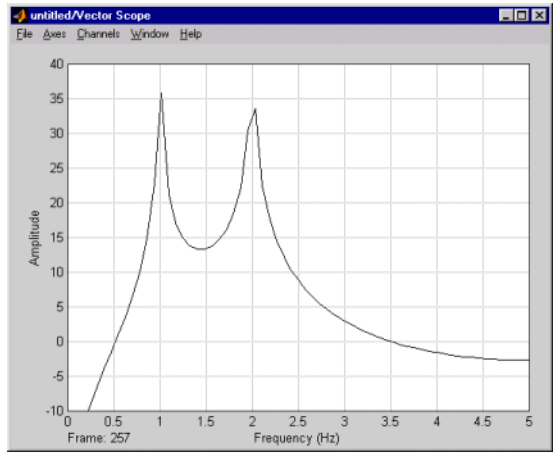
The frame period of the resulting summed sinusoid is 12.8 seconds (i.e., 128∗0.1), which is confirmed by the Probe block when the model is updated.

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

Configure the Vector Scope block as follows:

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

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

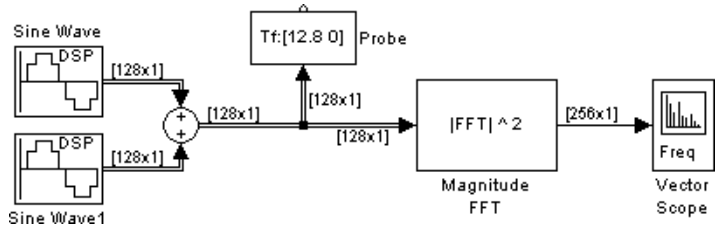


The Vector Scope block uses the input frame size (128) and period (12.8) to deduce the original signal's sample period (0.1), which allows it to *correctly* display the peaks at 1 Hz and 2 Hz.

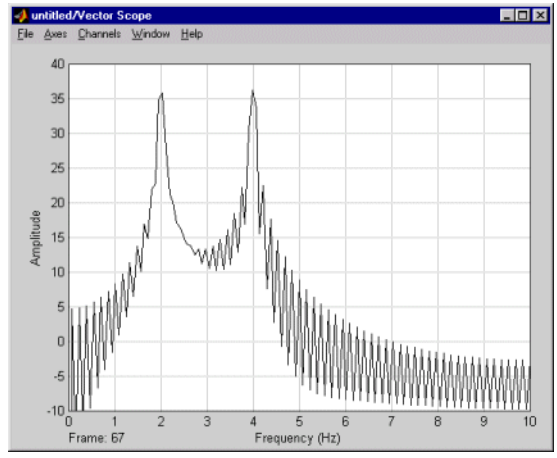
Example 2: Unintended Rate Conversion. Now alter the previous example by setting the Magnitude FFT block parameters as follows:

- Deselect 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 seconds (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 the following sections for more information about these methods:

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

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

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

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

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

Creating Signals

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

- "Creating Signals Using Constant Blocks"
- "Creating Signals Using Signal Generator Blocks"
- "Creating Signals Using the Signal From Workspace Block"

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

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

Creating Signals Using Constant Blocks

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

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

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

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

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

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

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

For information about importing signals, see the following sections:

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

Example: Creating Signals with the DSP Constant Block

The DSP Constant block has the following parameters:

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

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

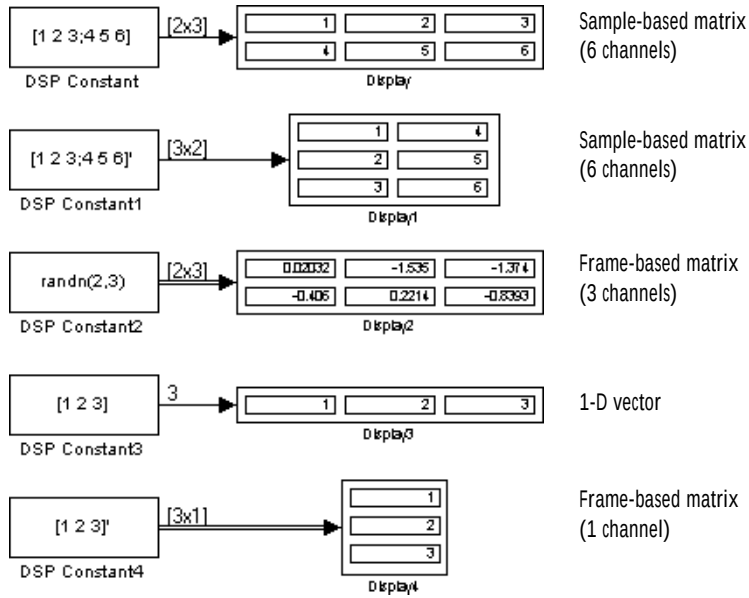
```
[1 2 3;4 5 6]      % A 2-by-3 matrix

[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 non-default settings for the other parameters: The third block (DSP Constant2) has the **Frame-based**

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



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

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

block is selected. This means that the output *is not a matrix*. However, most nonsource DSP blocks interpret a length-M 1-D vector as an M-by-1 matrix (column vector).

Note A 1-D vector signal must always be sample-based. The **Interpret vector parameters as 1-D** parameter is ignored when **Frame-based output** is selected, or when a matrix is specified for the **Constant value** parameter.

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

Creating Signals Using Signal Generator Blocks

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

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

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

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

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

Example: Creating Signals with the Sine Wave Block

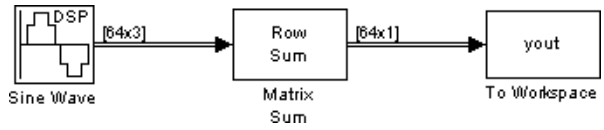
The Sine Wave block dialog box contains the following key parameters.

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

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

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

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



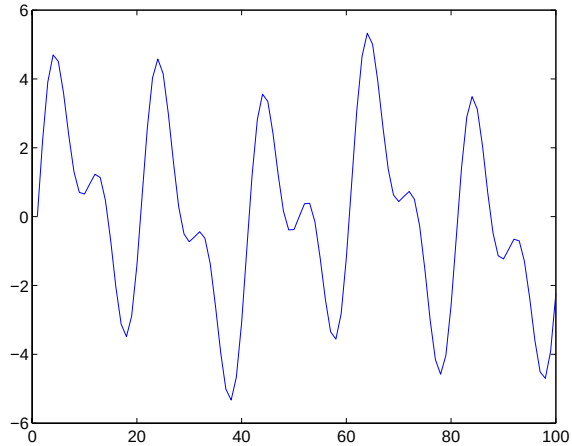
To build the model, set the **Sum along** parameter of the Matrix Sum block to **Rows**, and make the following parameter settings in the Sine Wave block:

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

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

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

at the command line.



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

Creating Signals Using the Signal From Workspace Block

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

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

- “Example 1: Generating Sample-Based Output”
- “Example 2: Generating Frame-Based Output”

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

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

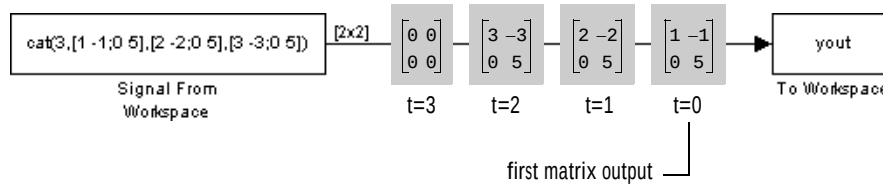
For more information about creating signals, see the following sections:

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

Example 1: Generating Sample-Based Output

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

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



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

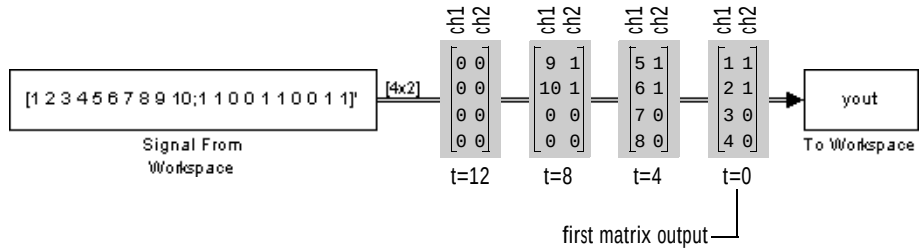
- **Signal** = `cat(3,[1 -1;0 5],[2 -2;0 5],[3 -3;0 5])`
- **Sample time** = `1`
- **Samples per frame** = `1`
- **Form output after final data value** = **Setting to zero**

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

Example 2: Generating Frame-Based Output

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

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



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

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

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

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

```
yout =

     1     1
     2     1
     3     0
     4     0
     5     1
     6     1
     7     0
     8     0
     9     1
    10     1
```

```
0     0
0     0
```

Constructing Signals

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

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

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

- "Constructing Multichannel Sample-Based Signals"
- "Constructing Multichannel Frame-Based Signals"

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

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

Constructing Multichannel Sample-Based Signals

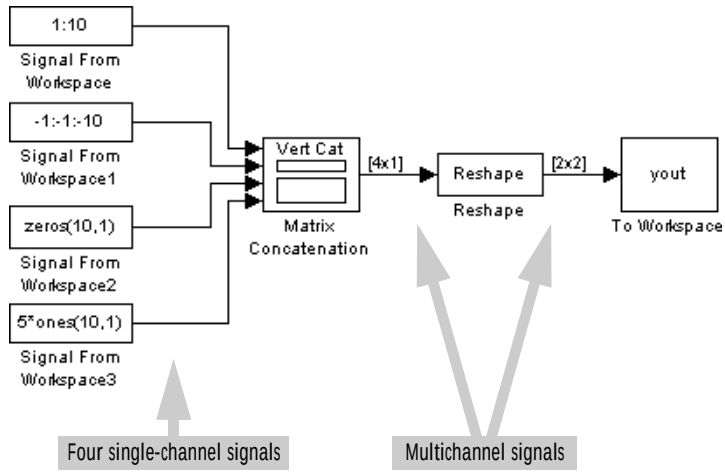
A sample-based signal with M∗N channels is represented by a sequence of M-by-N matrices. (The special case of M = N = 1 represents a single-channel signal.) Multiple individual signals can be combined into a multichannel matrix signal using the Matrix Concatenation block. Individual signals can be added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

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

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 Simulink's Sources library.

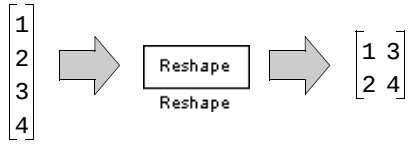
Example: Concatenating Single-Channel Signals. In the model below, four independent sample-based signals are combined into a 2-by-2 multichannel matrix signal.



To build the model, make the following parameter settings:

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

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

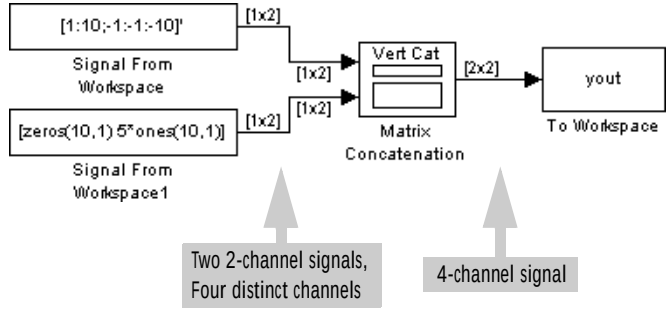


The 4-by-1 matrix and the 2-by-2 matrix in the above model represent the same sample-based four-channel signal. In some cases one representation may be more useful than the other. See “Sample-Based Multichannel Signals” on page 3-11 for more about sample-based signals.

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

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

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



To build the model, make the following parameter settings:

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

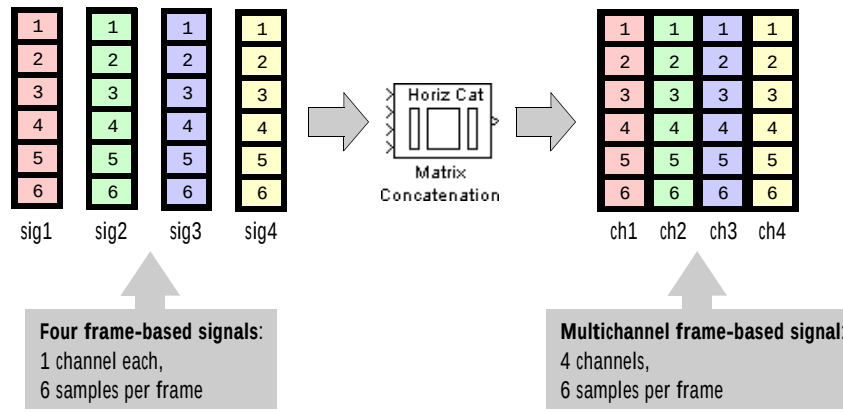
- In Signal From Workspace1, set **Signal** = `[zeros(10,1) 5*ones(10,1)]`
- In Matrix Concatenation, set:
  - **Number of inputs** = 2
  - **Concatenation method** = **Vertical**

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

Constructing Multichannel Frame-Based Signals

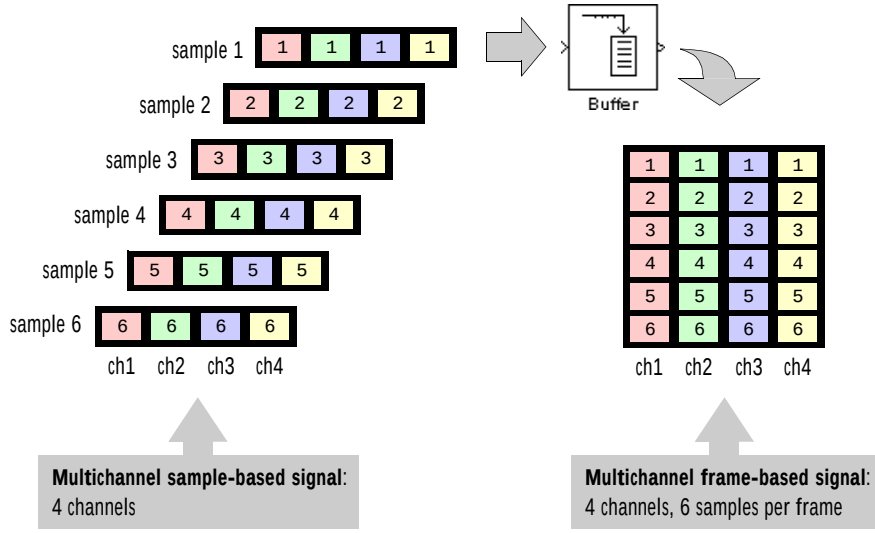
A frame-based signal with N channels and frame size M is represented by a sequence of M-by-N matrices. (The special case of N = 1 represents a single-channel signal.) There are two basic ways to construct a multichannel frame-based signal from existing signals:

- *By horizontally concatenating existing frame-based signals* – Multiple individual frame-based signals (with the same frame rate and size) can be combined into a multichannel frame-based signal using the Simulink Matrix Concatenation block. Individual signals can be added to an existing multichannel signal in the same way.



- *By buffering existing sample-based or frame-based signals* – Multichannel sample-based and frame-based signals can be buffered into multichannel

frame-based signals using the Buffer block in the Buffers library (in Signal Management).



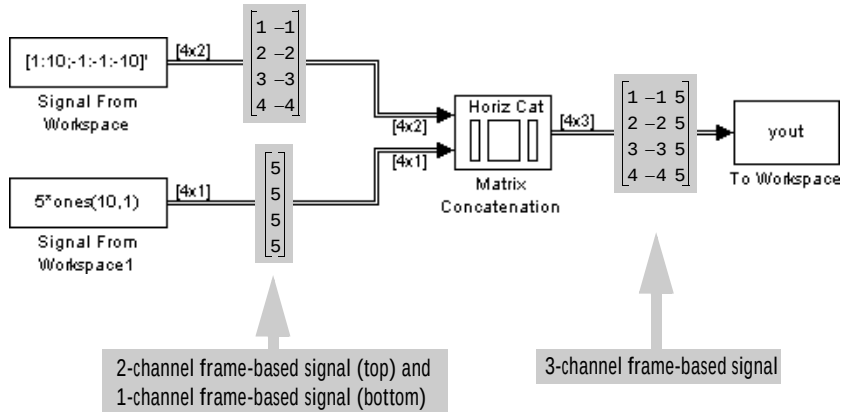
The following sections explain the two methods of constructing multichannel frame-based signals:

- "Concatenating Independent Frame-Based Signals into Multichannel Signals"
- "Buffering Sample-Based and Frame-Based Signals"

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 Simulink's Sources library. All signals must have the same frame rate and frame size.

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



To build the model, make the following parameter settings:

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

The 4-by-3 matrix output from the Matrix Concatenation block contains all three input channels, and preserves their common frame rate and frame size. See “Frame-Based Multichannel Signals” on page 3-12 for more about frame-based signals.

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

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

Buffering Sample-Based and Frame-Based Signals

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

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

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

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

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

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

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

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

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

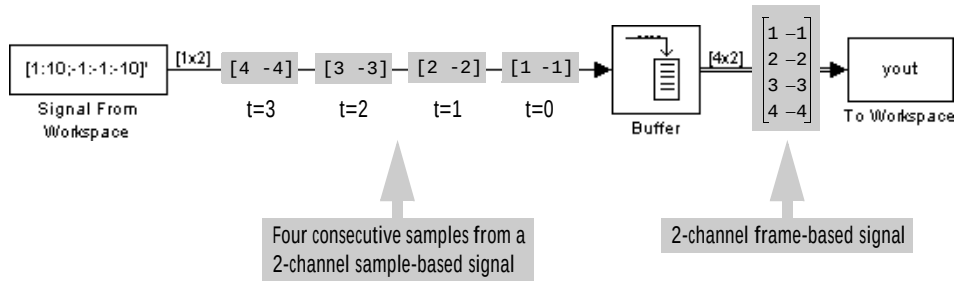
Clearly, this is equal to the input sample period *only* when the **Buffer overlap** is zero. See “Converting Sample Rates and Frame Rates” on page 3-20 for more information about rate conversions.

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

- “Example: Buffering Sample-Based Signals without Overlap”
- “Overlapping Buffers”
- “Example: Buffering Sample-Based Signals with Overlap”

- "Example: Buffering Frame-Based Signals with Overlap"
- "Buffering Delay and Initial Conditions"

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



To build the model, make the following parameter settings:

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

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

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

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

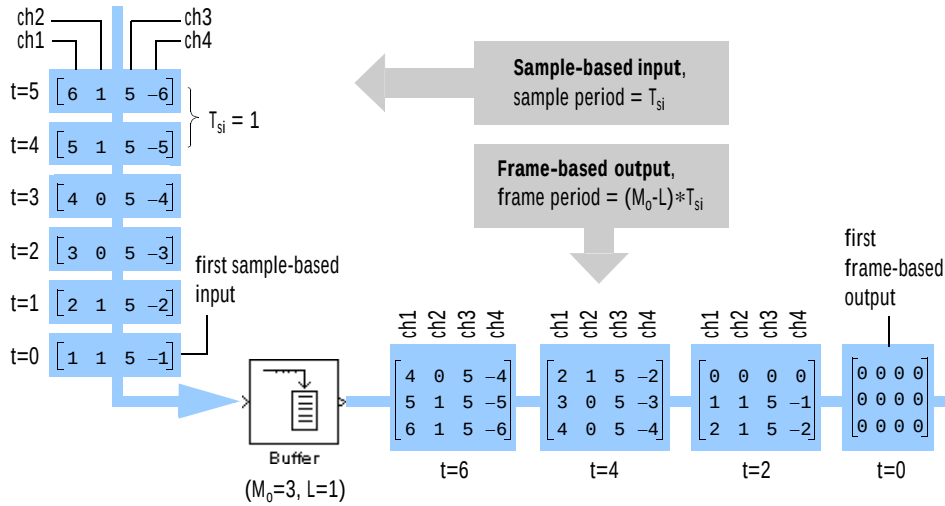
• "Importing a Multichannel Frame-Based Signal" on page 3-68

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

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

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

Example: Buffering Sample-Based Signals with Overlap. In the following model, a four-channel sample-based signal with sample period 1 is buffered to a frame-based signal with frame size 3 and frame period 2. Because of the overlap, the input sample period is not conserved, and the output sample period is 2/3.



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

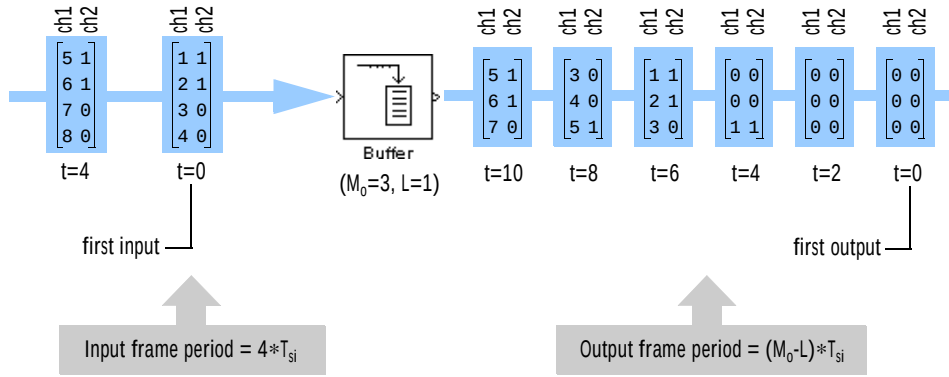
```
A = [1 1 5 -1;2 1 5 -2;3 0 5 -3;4 0 5 -4;5 1 5 -5;6 1 5 -6];
```

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

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

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

Example: Buffering Frame-Based Signals with Overlap. In the model below, a two-channel frame-based signal with frame period 4 is rebuffered to a frame-based signal with frame size 3 and frame period 2. Because of the overlap, the input sample period is not conserved, and the output sample period is 2/3.



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

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

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

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

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

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

In fact, under most conditions the Buffer and Unbuffer blocks have some amount of *latency*. This latency depends on both the block parameter settings and Simulink's tasking mode. You can use the `rebuffer_delay` function to determine the length of the block's latency for any combination of frame size and overlap.

The syntax `rebuffer_delay(f,n,m)` returns the delay (in samples) introduced by the buffering and unbuffering blocks in multitasking operations, where `f` is the input frame size, `n` is the **Buffer size** parameter setting, and `m` is the **Buffer overlap** parameter setting.

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

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

d =
     8
```

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

Deconstructing Signals

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

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

- "Deconstructing Multichannel Sample-Based Signals"
- "Deconstructing Multichannel Frame-Based Signals"

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

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

Deconstructing Multichannel Sample-Based Signals

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

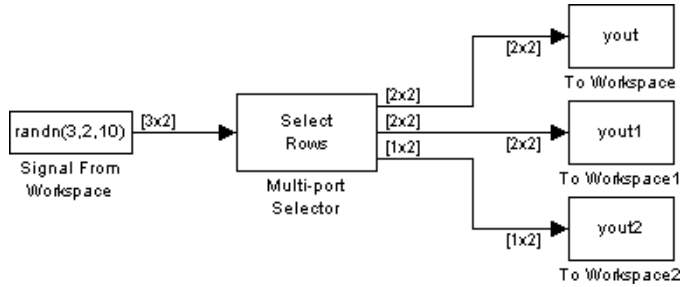
- "Deconstructing a Sample-Based Multichannel Signal into Multiple Independent Signals"
- "Deconstructing a Sample-Based Multichannel Signal into a Related Multichannel Signal"

Deconstructing a Sample-Based Multichannel Signal into Multiple Independent Signals

You can split a multichannel sample-based signal into individual sample-based signals (single-channel or multichannel) by using the Multiport Selector block

in the Indexing library (in Signal Management). Any subset of rows or columns can be selected for propagation to a given output port.

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



To build the model, make the following parameter settings:

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

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

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

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

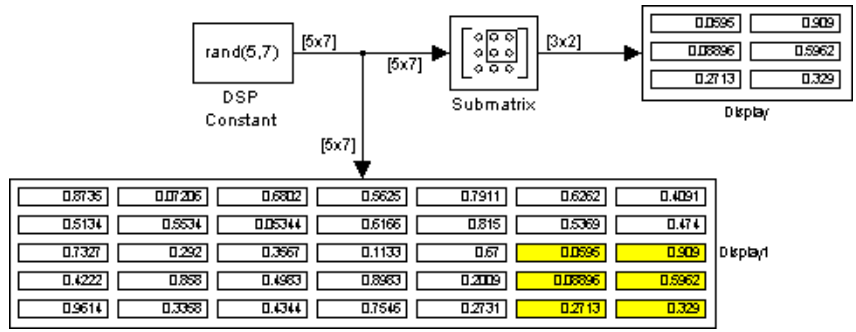
You can select a subset of channels from a multichannel sample-based signal by using one of the following blocks in the Indexing library (in Signal Management):

- Selector (Simulink)
- Submatrix

- Variable Selector

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

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



To build the model, make the following parameter settings:

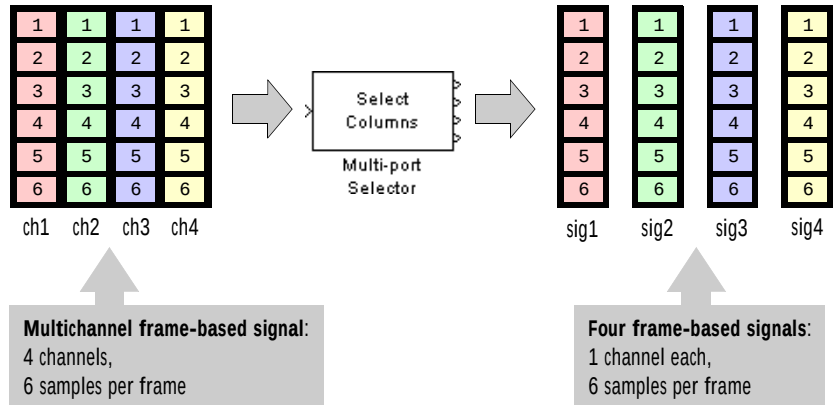
- In DSP Constant, set **Constant value** = `rand(5,7)`
- In Submatrix, set:
  - **Row span** = **Range of rows**
  - **Starting row** = **Index**
  - **Starting row index** = `3`
  - **Ending row** = **Last**
  - **Column span** = **Range of columns**
  - **Starting column** = **Offset from last**
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

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

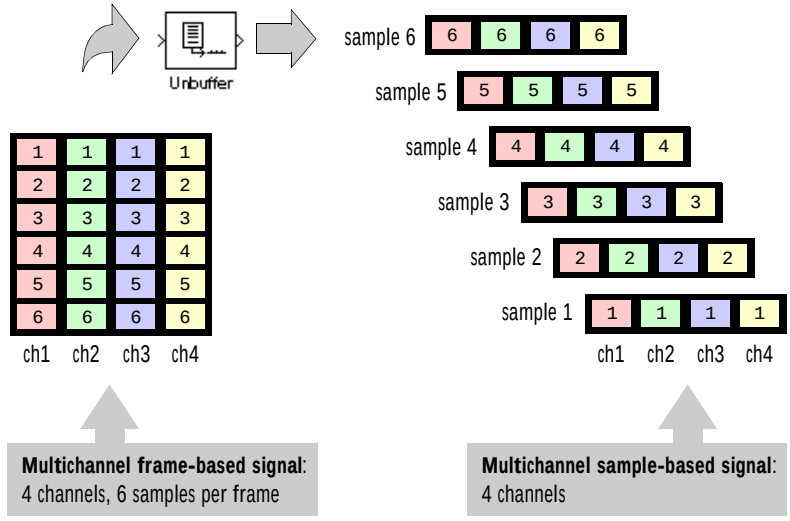
Deconstructing Multichannel Frame-Based Signals

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

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



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



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

- "Splitting a Multichannel Signal into Individual Signals"
- "Unbuffering a Frame-Based Signal into a Sample-Based Signal"

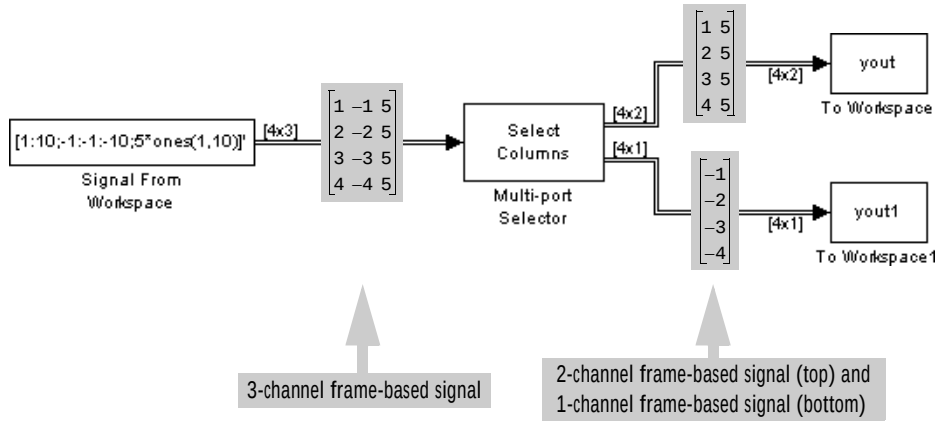
The final section explains how to reorder the channels in a frame-based signal *without* splitting the channels apart:

- "Reordering Channels in a Frame-Based Multichannel Signal"

Splitting a Multichannel Signal into Individual Signals

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

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



To build the model, make the following parameter settings:

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

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

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

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

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

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

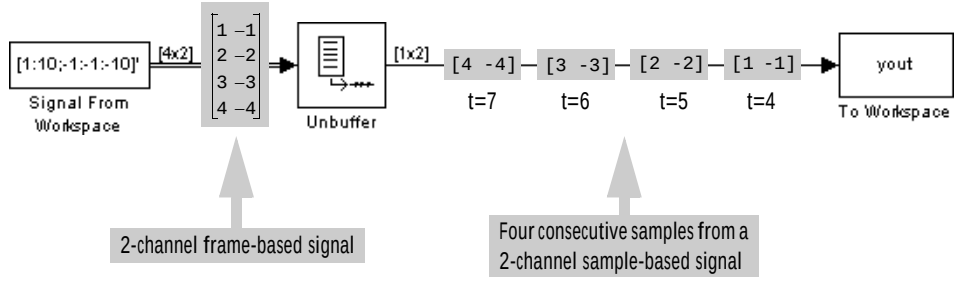
The Unbuffer block performs the inverse operation of the Buffer block's "sample-based to frame-based" buffering process, and generates an N-channel sample-based output from an N-channel frame-based input. The first row in each input matrix is always the first sample-based output. In other words, the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

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

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

The Unbuffer block always preserves the signal's sample period ($T_{so} = T_{si}$). See "Converting Sample Rates and Frame Rates" on page 3-20 for more information about rate conversions.

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



To build the model, make the following parameter settings:

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

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

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

Reordering Channels in a Frame-Based Multichannel Signal

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

Importing Signals

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

- "Importing a Multichannel Sample-Based Signal"
- "Importing a Multichannel Frame-Based Signal"
- "Importing WAV Files"

For information about creating signals, see the following sections:

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

Importing a Multichannel Sample-Based Signal

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

The dialog box has the following parameters:

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

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

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

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

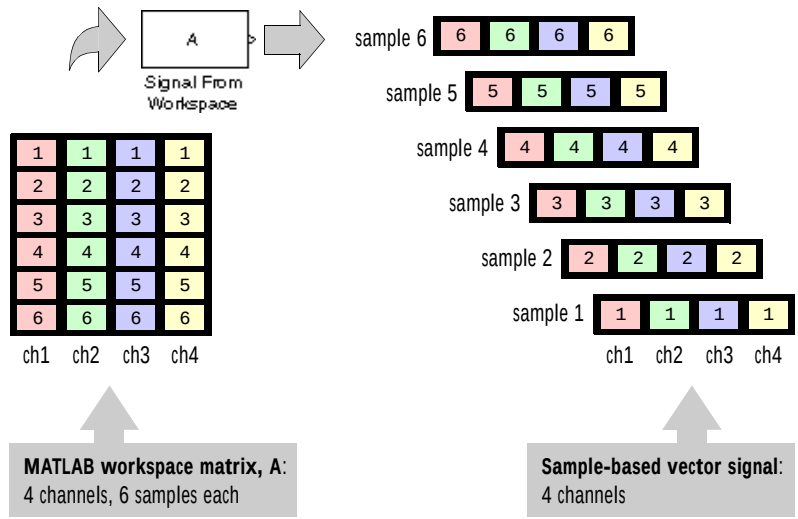
The following sections explain how the Signal From Workspace generates its output:

- “Importing a Sample-Based Vector Signal”
- “Importing a Sample-Based Matrix Signal”

Importing a Sample-Based Vector Signal

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

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



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

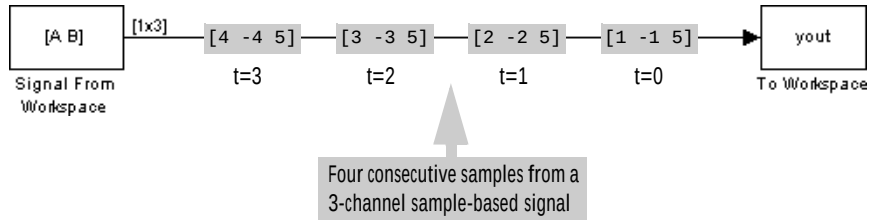
containing M consecutive samples. In other words, the workspace matrix must be oriented so as to have the independent channels as its columns.

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

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

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

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



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

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

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

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

- **Signal** = `[A B]`
- **Sample time** = `1`
- **Samples per frame** = `1`

- **Form output after final data value** = **Setting to zero**

The **Signal** expression `[A B]` uses the standard MATLAB syntax for horizontally concatenating matrices and appends column vector `B` to the right of matrix `A`. Equivalently, you could set **Signal** = `C`, and define `C` at the command line by

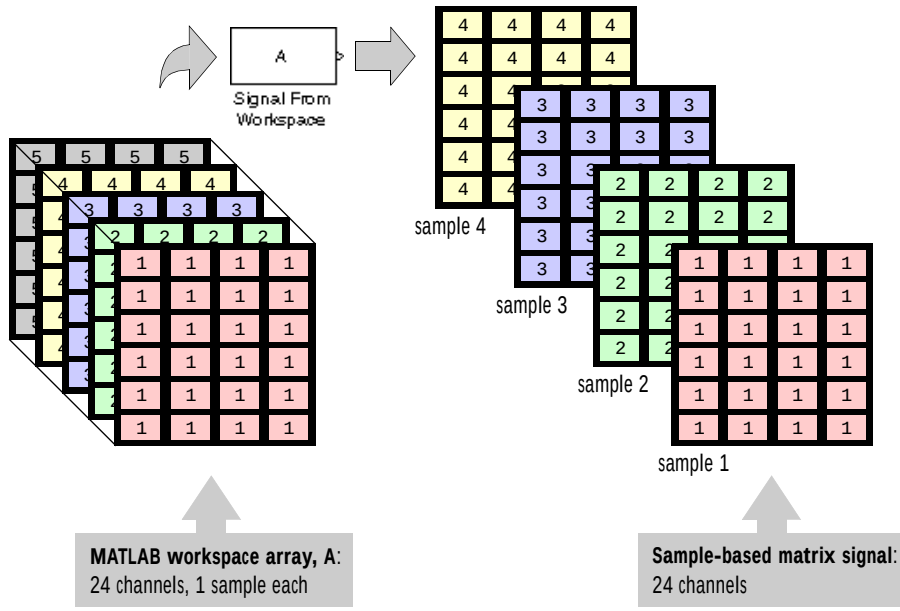
```
C = [A B]
```

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

Importing a Sample-Based Matrix Signal

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

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



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

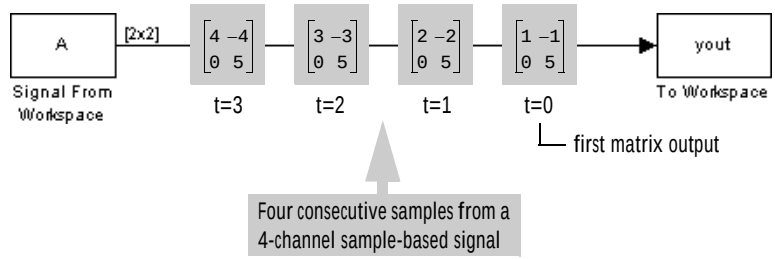
When the block has output all of the pages available in the specified array, it can start again at the beginning of the array, or simply repeat the final page (or generate zero-matrices) until the end of the simulation. This behavior is controlled by the **Form output after final data value by** parameter.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

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

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

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

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



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

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

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

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

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

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

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

The following two sections may also be of interest:

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

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

Importing a Multichannel Frame-Based Signal

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

The dialog box has the following parameters:

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

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

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

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

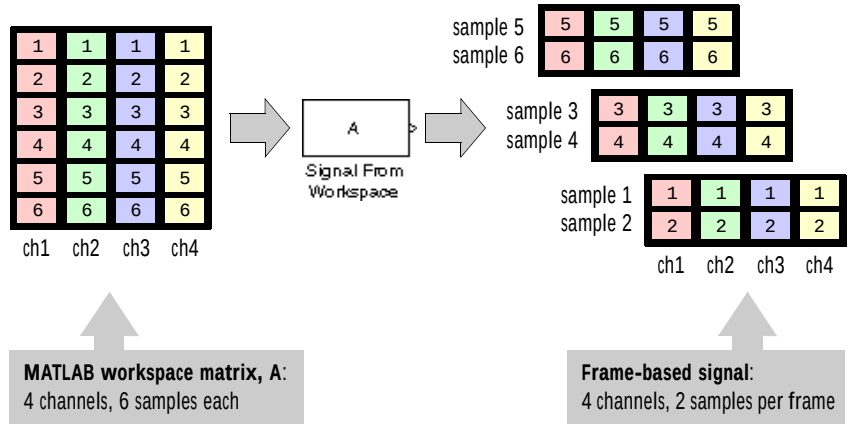
The following section explains how the Signal From Workspace generates its frame-based output.

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

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

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

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



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

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

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

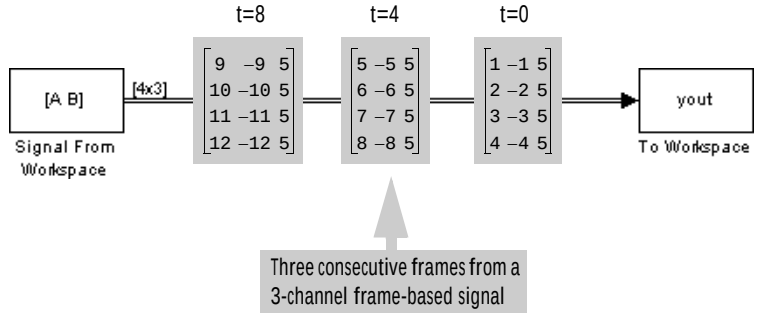
by the **Form output after final data value by** parameter. See the Signal From Workspace reference page for more information.

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 creates a three-channel frame-based signal with the following data:

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

The frame size is four samples.



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

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

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

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

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

The **Signal** expression `[A B]` uses the standard MATLAB syntax for horizontally concatenating matrices and appends column vector `B` to the right of matrix `A`. Equivalently, you could set **Signal** = `C`, and define `C` at the command line by

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

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

Importing WAV Files

The key blocks for importing WAV audio files are:

- From Wave Device
- From Wave File

See the reference pages for complete information.

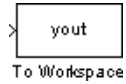
Exporting Signals

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"
- "Exporting and Playing WAV Files"

Exporting Multichannel Signals

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



The dialog box has the following parameters:

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

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

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

The default values of `1` and `-1` for the **Decimation** and **Sample time** parameters (respectively) are generally adequate for DSP models. If you want

to downsample a signal before exporting to the workspace, consider using the Downsample or FIR Decimation blocks. See "Converting Sample Rates and Frame Rates" on page 3-20 for more information about rate conversion.

The **Save format** parameter allows you to save the output in a variety of formats. The default is **Array**, which is also generally the most accessible output format. Although this format does not save a record of the sample times corresponding to the output samples, you can create such a record for a given model by selecting the **Time** option in the **Workspace I/O** panel of the **Simulation Parameters** dialog box. See "Performance-Related Settings" on page 2-13 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"
- "Exporting a Frame-Based Signal Using the To Workspace Block"

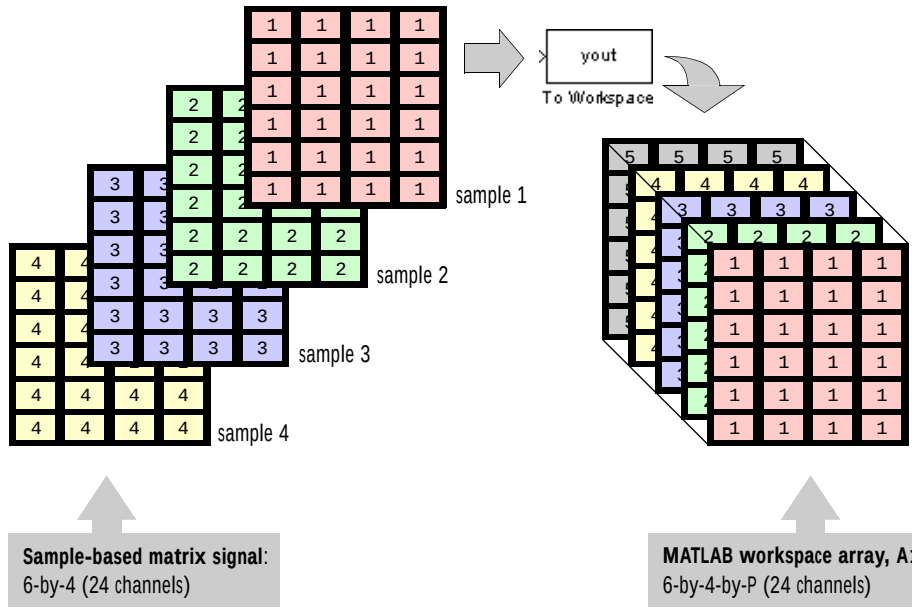
The following two sections may also be of interest:

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

Exporting a Sample-Based Signal Using the To Workspace Block

Recall that a sample-based signal with M∗N channels is represented by a sequence of M-by-N matrices. (The special case of M = N = 1 represents a single-channel signal.) When the input to the To Workspace block is a sample-based signal (and the **Save format** parameter is set to **Array**), the block creates an M-by-N-by-P array in the MATLAB workspace containing the P most recent samples from each channel. The number of pages, P, is specified by the **Limit data points to last** parameter. The newest samples are added at the back of the array.

The figure below illustrates this for a 6-by-4 sample-based signal exported to workspace array `A`.



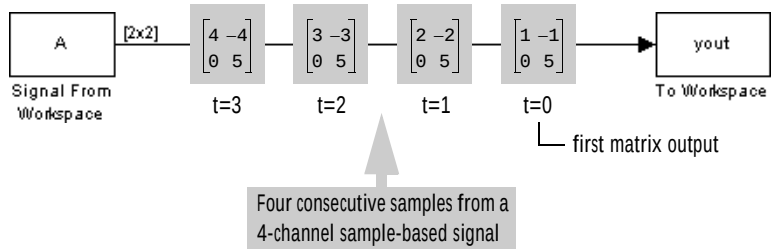
The workspace array always has time running along its third (P) dimension. Samples are saved along the P dimension whether the input is a matrix, vector, or scalar (single channel).

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

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

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

The first four consecutive samples are shown in the figure.



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

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

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

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

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

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

Specify the following parameter values in the To Workspace block:

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

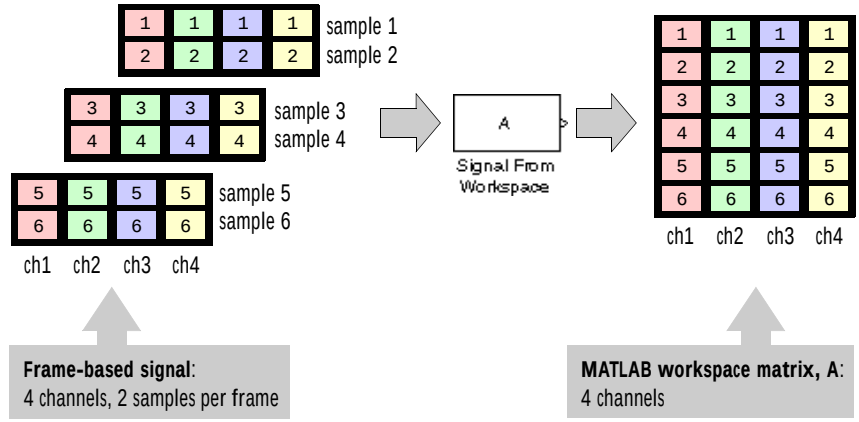
Run the model, and look at output `yout`. The first four samples (pages) are shown below.

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

ans(:,:,1) =

     1    -1
     0     5

ans(:,:,2) =

     2    -2
     0     5

ans(:,:,3) =

     3    -3
     0     5

ans(:,:,4) =

     4    -4
     0     5
```

Exporting a Frame-Based Signal Using the To Workspace Block

Recall that a frame-based signal with N channels and frame size M is represented by a sequence of M-by-N matrices. (The special case of N = 1 represents a single-channel signal.) When the input to the To Workspace block is a frame-based signal (and the **Save format** parameter is set to **Array**), the block creates an P-by-N array in the MATLAB workspace containing the P most recent samples from each channel. The number of rows, P, is specified by the **Limit data points to last** parameter. The newest samples are added at the bottom of the matrix.

The figure below illustrates this for three consecutive frames of a frame-based signal (two samples per frame) exported to matrix A.

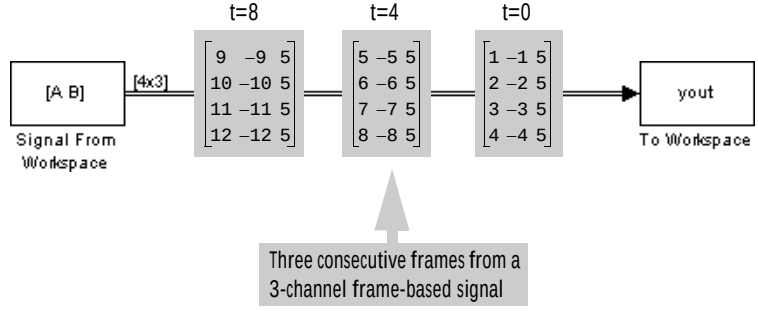


The workspace matrix always has time running along its first (P) dimension. Samples are saved along the P dimension whether the input is a matrix, vector, or scalar (single channel).

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

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

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



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

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

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

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

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

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

Specify the following parameter values in the To Workspace block:

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

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

```
yout =

     1    -1     5
     2    -2     5
     3    -3     5
     4    -4     5
     5    -5     5
     6    -6     5
     7    -7     5
     8    -8     5
     9    -9     5
    10   -10     5
```

The following two sections may also be of interest:

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

Exporting and Playing WAV Files

The key blocks for exporting and playing WAV audio files are:

- To Wave Device
- To Wave File

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

The following demos may also be of interest:

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

Viewing Signals

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

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

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

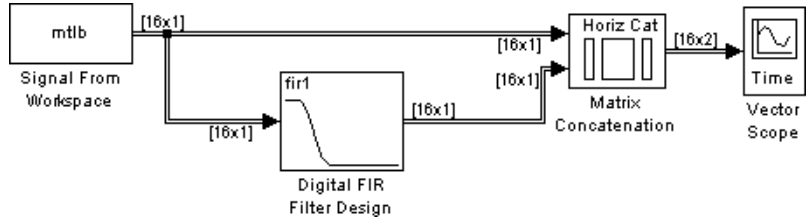
- "Displaying Signals in the Time-Domain"
- "Displaying Signals in the Frequency-Domain"
- "Displaying Matrices"

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 FIR Filter Design block:

- **Filter type** = **Lowpass**
- **Filter order** = `22`
- **Cutoff frequency** = `0.25`
- **Window type** = **Hamming**

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.

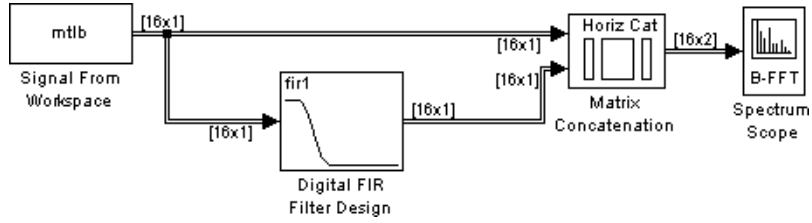
- Select **CH 1** from the right-click menu, and then select **Color** and **Blue** from the submenus, to code the channel 1 signal with the color blue.
- Select **CH 2** from the right-click menu, and then select **Marker** and **Diamond** from the submenus, to mark the data points on the channel 2 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 FIR Filter Design block:

- **Filter type** = **Lowpass**
- **Filter order** = `22`
- **Cutoff frequency** = `0.25`

- **Window type** = **Hamming**

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

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

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

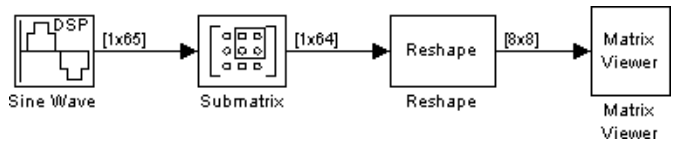
You may want to set the **Stop time** in the **Simulation Parameters** dialog box to `inf` to allow the simulation to run longer. See "Improving the Appearance of the Display" on page 3-81 for some tips on improving the scope display.

Displaying Matrices

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

Example: Displaying Matrices

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



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

- **Amplitude** = `1`

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

Specify the following parameter values in the Submatrix block:

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

Specify the following parameter values in the Reshape block:

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

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

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

Delay and Latency

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

- Computational delay
- Algorithmic delay

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 Simulink's Digital Clock block) does not provide any information about computational delay. For example, according to the Simulink timer, the FFT mentioned above executes instantaneously, with no delay whatsoever. An input to the FFT block at simulation time t=25.0 is processed and output at time t=25.0, regardless of the number of operations performed by the FFT algorithm. The Simulink timer reflects only algorithmic delay (described below), not computational delay.

The next section discussed methods of reducing computational delay.

Reducing Computational Delay

There are a number of ways to reduce computational delay without actually running the simulation on faster hardware. To begin with, you should familiarize yourself with "Improving Simulation Performance and Accuracy" in the Simulink documentation, which describes some basic strategies. The section below supplements that information with several additional options for improving performance.

A first step in improving performance is to analyze your model, and eliminate or simplify elements that are adding excessively to the computational load. Such elements might include scope displays and data logging blocks that you had put in place for debugging purposes and no longer require. In addition to these model-specific adjustments, there are a number of more general steps you can take to improve the performance of any model:

- Use frame-based processing wherever possible. It is advantageous for the entire model to be frame-based. See "Benefits of Frame-Based Processing" on page 3-14 for more information.
- Use the `dspstartup` file to tailor Simulink for DSP models, or manually make the adjustments described in "Performance-Related Settings" on page 2-13.
- 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 launching a simulation can greatly increase the simulation speed, but also has several limitations:
  - You cannot interact with the simulation (to tune parameters, for instance).
  - You must press **Ctrl+C** to stop the simulation, or specify start and stop times.
  - There are no graphics updates in M-file S-functions, which include blocks such as the frame scopes (Vector Scope, etc.).
- Use the Real-Time Workshop to generate generic real-time (GRT) code targeted to your host platform, and simulate the model using the generated executable file. See the Real-Time Workshop documentation for more information.

Algorithmic Delay

Algorithmic delay is delay that is intrinsic to the algorithm of a block or subsystem, and is independent of CPU speed. In Chapter 5, "DSP Block Reference," and elsewhere in this guide, the algorithmic delay of a block is referred to simply as the block's *delay*. It is generally expressed in terms of the number of samples by which a block's output lags behind the corresponding

input. This delay is directly related to the time elapsed on the Simulink timer during that block's execution.

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

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

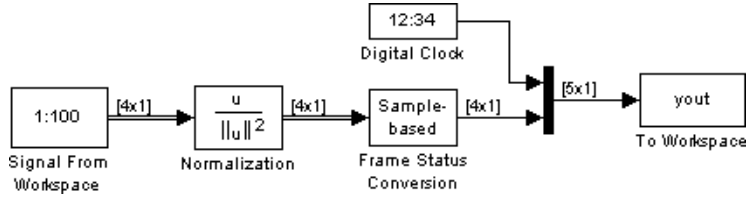
The following sections explain the different levels of delay, and how the simulation and parameter settings can affect the level of delay that a particular block experiences.

Zero Algorithmic Delay

The FFT block is an example of a component that has *no* algorithmic delay; the Simulink timer does not record any passage of time while the block computes the FFT of the input, and the transformed data is available at the output in the same time step that the input is received. There are many other blocks that have zero algorithmic delay, such as the blocks in the Matrices and Linear Algebra libraries. Each of those blocks processes its input and generates its output in a single time step.

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

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



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

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

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

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

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

```
(t=0)   [ 1  2  3  4]'
(t=1)   [ 5  6  7  8]'
(t=2)   [ 9 10 11 12]'
(t=3)   [13 14 15 16]'
(t=4)   [17 18 19 20]'
```

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

```
squeeze(yout)'
```

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

```
ans =

         0    0.0333    0.0667    0.1000    0.1333
    1.0000    0.0287    0.0345    0.0402    0.0460
    2.0000    0.0202    0.0224    0.0247    0.0269
    3.0000    0.0154    0.0165    0.0177    0.0189
    4.0000    0.0124    0.0131    0.0138    0.0146
      time
```

The first column of ans is the Simulink time provided by the Digital Clock block. You can see that the squared 2-norm of the first input,

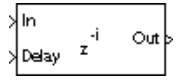
```
[1 2 3 4]' ./ sum([1 2 3 4]'.^2)
```

appears in the first row of the output (at time *t*=0), the same time step that the input was received by the block. This indicates that the Normalization block has zero algorithmic delay.

Zero Algorithmic Delay and Algebraic Loops. When several blocks with zero algorithmic delay are connected in a feedback loop, Simulink may report an *algebraic loop error* and performance may generally suffer. You can prevent algebraic loops by injecting at least one sample of delay into a feedback loop (for example, by including an Integer Delay block with **Delay** > 0). See the Simulink documentation for more information about algebraic loops.

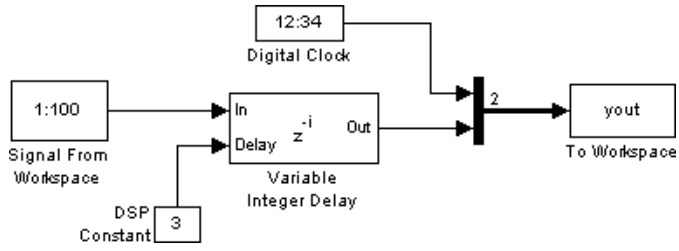
Basic Algorithmic Delay

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



The input to the Delay port of the block specifies the number of sample periods that should elapse before an input to the In port is released to the output. This value represents the block's algorithmic delay. For example, if the input to the Delay port is a constant 3, and the sample period at both ports is 1, then a sample that arrives at the block's In port at time *t*=0 is released to the output at time *t*=3.

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



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

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

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

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

```
yout =

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

The first column of `yout` is the Simulink time provided by the Digital Clock block, and the second column is the delayed input. As expected, the input to the block at t=0 is delayed three samples, and appears as the fourth output sample, at t=3. You can also see that the first three outputs from the Variable Integer Delay block inherit the value of the block's **Initial conditions** parameter, `-1`. This period of time, from the start of the simulation until the first input is propagated to the output, is sometimes called the *initial delay* of the block.

Many blocks in the DSP Blockset have some degree of fixed or adjustable algorithmic delay. These include any blocks whose algorithms rely on delay or storage elements, such as filters or buffers. Often (but not always), such blocks provide an **Initial conditions** parameter that allows you to specify the output values generated by the block during the initial delay. In other cases, the initial conditions are internally fixed at 0.

Consult Chapter 5, "DSP Block Reference," 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 Simulink's 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 Simulink's 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.

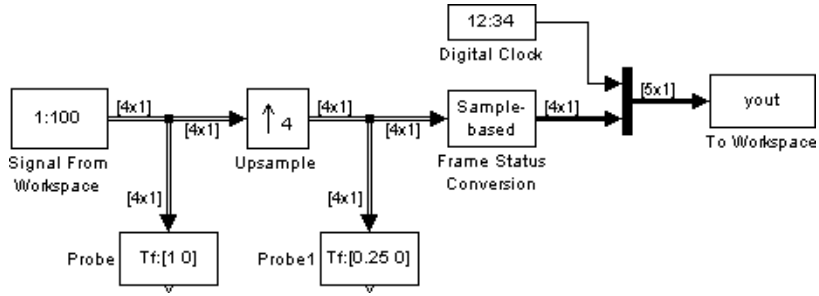
Block Sample Mode. Many blocks can operate in either sample-based or frame-based modes. In source blocks, the mode is usually determined by the **Samples per frame** parameter; a value of `1` for this parameter indicates sample-based mode, while a value greater than `1` indicates frame-based mode. In nonsource blocks, the sample mode is determined by the input signal. See Chapter 5, "DSP 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 Chapter 5, "DSP Block Reference," to predict tasking latency:

- "Example: Nonzero Tasking Latency"
- "Example: Zero Tasking Latency"

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



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

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

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

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

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

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

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

```
(t=0)   [ 1  2  3  4]
(t=1)   [ 5  6  7  8]
(t=2)   [ 9 10 11 12]
(t=3)   [13 14 15 16]
(t=4)   [17 18 19 20]
```

The Upsample block upsamples the input by a factor of 4, inserting three zeros between each input sample. The change in rates is confirmed by the Probe blocks in the model, which show a decrease in the frame period from T_{fi} = 1 to T_{fo} = 0.25.

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

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

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

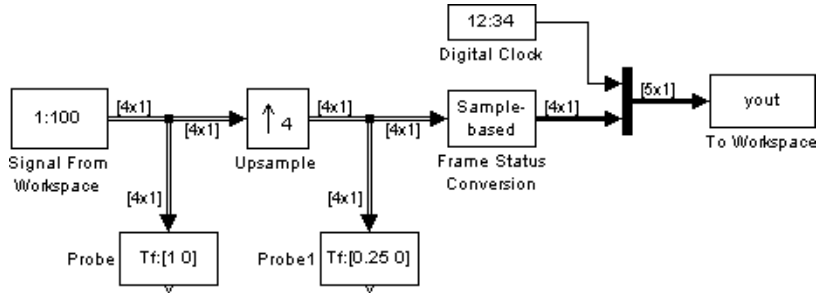
```
squeeze(yout)'
```

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

```
ans =

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

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

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



To engage Simulink's 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 Simulink’s single-tasking mode. To verify this, look at the output from the simulation, `squeeze(yout)'`. The first few samples are shown below.

```
ans =
```

time				
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

1st output frame

5th output frame

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 Simulink’s single-tasking mode therefore eliminates the 17-sample delay that the model experiences under Simulink’s multitasking mode (for the particular parameter settings in the example).

4

DSP Operations

Overview

This chapter discusses some basic DSP operations, and how they can be implemented using the DSP Blockset. The following topics are covered:

- "Filters"
- "Transforms"
- "Power Spectrum Estimation"
- "Linear Algebra"
- "Statistics"

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

A final section, "DSP Blockset Demos Overview" on page 4-39, provides a look at the demonstration models that accompany the DSP Blockset.

Filters

Filtering is one of the most important operations in signal processing, and is supported in the DSP Blockset with four libraries of filtering blocks. The following sections present a brief overview of these libraries:

- "Adaptive Filters"
- "Filter Designs"
- "Filter Structures"
- "Multirate Filters"

All libraries are located within the top-level Filtering library.

Adaptive Filters

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

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

The DSP Blockset provides a collection of adaptive filtering demos to illustrate how these blocks can be used:

- 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`)

Filter Designs

Because the chief purpose of a filter is to alter the frequency content of a signal, it is natural to describe filter characteristics in the frequency domain. The blocks in the Filter Designs library let you specify the characteristics of a filter using convenient frequency-domain criteria. You enter the desired filter

response characteristics, such as cutoff frequency and bandwidth, and the block automatically computes the filter coefficients that approximate the desired response. The filter is applied to the input signal using a direct-form II transpose filter structure, which yields the same results as the `filter` function in MATLAB. (For other architectures, see "Filter Structures" on page 4-23.)

The Filter Designs library contains the following seven blocks. The icon of each block displays the frequency response of the filter that is currently specified in the block's dialog box:

- Analog Filter Design
- Digital FIR Filter Design
- Digital FIR Raised Cosine Filter Design
- Digital IIR Filter Design
- Least Squares FIR Filter Design
- Remez FIR Filter Design
- Yule-Walker IIR Filter Design

The following sections provide further information about the capabilities of these blocks:

- "Filter Design Categorization"
- "Designing Discrete-Time Classical IIR and FIR Filters"
- "Designing Continuous-Time Classical IIR Filters"
- "Designing Discrete-Time Special IIR and FIR Filters"

All of the blocks in the Filter Designs library are built on the filter design capabilities of the Signal Processing Toolbox. For further details on any of the filter design topics discussed in the following sections, see the "Filter Designs" section of the Signal Processing Toolbox documentation.

Filter Design Categorization

The blocks in the Filter Designs library can be grouped into three categories:

- *Classical IIR and FIR Filters, Discrete Time*

  The Digital FIR Filter Design and Digital IIR Filter Design blocks design and implement classical discrete-time windowed filters with standard band configurations (highpass, lowpass, bandpass, or bandstop). The Digital FIR

Filter Design blocks additionally offers linear phase multiband and arbitrary shape configurations. The Digital IIR Filter Design block generates Butterworth, Chebyshev type I, Chebyshev type II, and elliptic designs.

See "Designing Discrete-Time Classical IIR and FIR Filters" below for more about these types of filters.

- *Classical IIR Filters, Continuous Time*

  The Analog Filter Design block designs and implements Butterworth, Chebyshev type I, Chebyshev type II, and elliptic filters in standard band configurations (highpass, lowpass, bandpass, or bandstop).

  See "Designing Continuous-Time Classical IIR Filters" on page 4-16 for more about these types of filters.

- *Designing Discrete-Time Special IIR and FIR Filters*

  The Remez FIR Filter Design, Yule-Walker IIR Filter Design, and Least Squares FIR Filter Design blocks design and implement IIR or FIR filters with arbitrary magnitude responses, including multiband responses. The Digital FIR Raised Cosine Filter Design block designs a discrete-time lowpass filter with a raised cosine transition region.

  See "Designing Discrete-Time Special IIR and FIR Filters" on page 4-17 for more about these types of filters.

	Analog	Digital FIR	Digital IIR	Remez FIR	Least Squares FIR	Yule-Walker IIR	Digital FIR Raised Cosine
Bands	Lowpass	Lowpass	Lowpass	Multiband	Multiband	Multiband	Lowpass
	Highpass	Highpass	Highpass				
	Bandpass	Bandpass	Bandpass				
	Bandstop	Bandstop	Bandstop				
		Multiband					
		Arbitrary					
Designs	Butterworth		Butterworth	Hilbert Trans	Hilbert Trans		
	Chebyshev I		Chebyshev I	Differentiator	Differentiator		
	Chebyshev II		Chebyshev II				
	Elliptic		Elliptic				

Designing Discrete-Time Classical IIR and FIR Filters

The Digital FIR Filter Design and Digital IIR Filter Design blocks primarily design and implement discrete-time filters with standard band configurations. The first two sections to follow describe the key design parameters available in

the block dialog boxes and how these parameters can be used to design a particular filter response:

- "Design Parameters of Classical Discrete-Time Filters"
- "Specifying Classical Discrete-Time Filter Parameters"

You may find it helpful to open the block dialog boxes on your computer, and experiment with the various parameter combinations as you read through these sections.

The next group of sections illustrate a variety of the responses that can be constructed using the block parameters:

- "Lowpass and Highpass Discrete-Time FIR Filters"
- "Bandpass and Bandstop Discrete-Time FIR Filters"
- "Multiband Discrete-Time FIR Filters"
- "Arbitrary Shape Discrete-Time FIR Filters"
- "Discrete-Time IIR Filters"

The final section, "Example: Chebyshev Type II Lowpass Filter" on page 4-14, presents a working lowpass filter model.

Design Parameters of Classical Discrete-Time Filters. All of the digital filter designs available in Digital FIR Filter Design and Digital IIR Filter Design blocks allow you to specify the order of the filter. In addition, depending on the specific design, you can specify one or more of the following frequency response parameters:

f_{n0} = normalized cutoff or band edge frequency
f_{n1} = normalized lower cutoff or band edge frequency
f_{n2} = normalized upper cutoff or band edge frequency
f_n = normalized cutoff or band edge frequency vector
m_n = normalized magnitude vector
R_p = passband ripple in decibels
R_s = stopband attenuation in decibels

The table below shows the possible combinations.

	FIR	Butterworth	Chebyshev I	Chebyshev II	Elliptic
Lowpass	f_{n0}	f_{n0}	f_{n0}, R_p	f_{n0}, R_s	f_{n0}, R_p, R_s
Highpass	f_{n0}	f_{n0}	f_{n0}, R_p	f_{n0}, R_s	f_{n0}, R_p, R_s
Bandpass	f_{n1}, f_{n2}	f_{n1}, f_{n2}	f_{n1}, f_{n2}, R_p	f_{n1}, f_{n2}, R_s	f_{n1}, f_{n2}, R_p, R_s
Bandstop	f_{n1}, f_{n2}	f_{n1}, f_{n2}	f_{n1}, f_{n2}, R_p	f_{n1}, f_{n2}, R_s	f_{n1}, f_{n2}, R_p, R_s
Multiband	f_n	n/a	n/a	n/a	n/a
Arbitrary Shape	f_n, m_n	n/a	n/a	n/a	n/a

The frequency response parameters available for the Digital FIR Filter Design block are determined by the option selected in the **Filter type** pop-up menu: **Lowpass**, **Highpass**, **Bandpass**, **Bandstop**, **Multiband**, or **Arbitrary shape**. The first four options are shared with the Digital IIR Filter Design block. The block dialog box adapts to show you the appropriate design parameters for whichever band configuration you select in the **Filter type** menu.

The four right columns in the above table show the various options presented in the Digital IIR Filter Design block's **Design method** pop-up menu: **Butterworth**, **Chebyshev I**, **Chebyshev II**, and **Elliptic**. Each column lists the filter specifications available in that design method for any band configuration that can be selected from the **Filter type** menu.

Specifying Classical Discrete-Time Filter Parameters. For both the Digital FIR Filter Design and Digital IIR Filter Design blocks, frequency parameters are normalized to half the sample frequency, so the cutoff or band edge frequencies are always in the range `[0 1]`. The magnitude frequency response is normalized to 1 (0 dB).

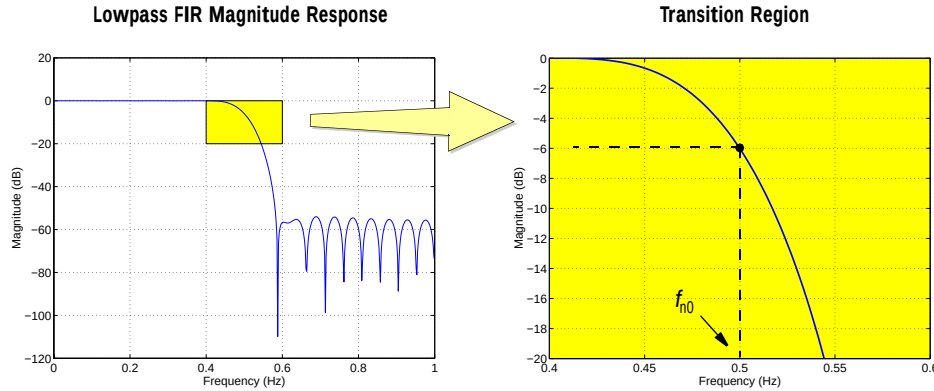
The FIR filters are real, symmetric, linear phase filters; Type I for even filter orders and Type II for odd filter orders. *Cutoff frequencies* are defined as the frequencies where the magnitude response drops to -6 dB below the passband level (one half the passband gain). *Band edge frequencies* are defined as the left or right edges of a transition region between bands.

The following four sections provide several examples of how cutoff and band edge parameters are used to specify FIR filter responses:

- “Lowpass and Highpass Discrete-Time FIR Filters”
- “Bandpass and Bandstop Discrete-Time FIR Filters”
- “Multiband Discrete-Time FIR Filters”
- “Arbitrary Shape Discrete-Time FIR Filters”

The final section, “Discrete-Time IIR Filters” on page 4-13, illustrates how cutoff and band edge parameters are used to specify an IIR filter response.

Lowpass and Highpass Discrete-Time FIR Filters. The magnitude response illustrated below shows a lowpass order-41 FIR filter with normalized cutoff frequency f_{n0} = 0.5. The right plot shows a zoomed view of the transition region. For an input signal sampled at 1 kHz, this filter would provide a cutoff frequency of 250 Hz.



The equivalent MATLAB code for this filter design is

```
b = fir1(41,0.5)
```

and the filter can be designed in the Digital FIR Filter Design block by making the following selections in the dialog box:

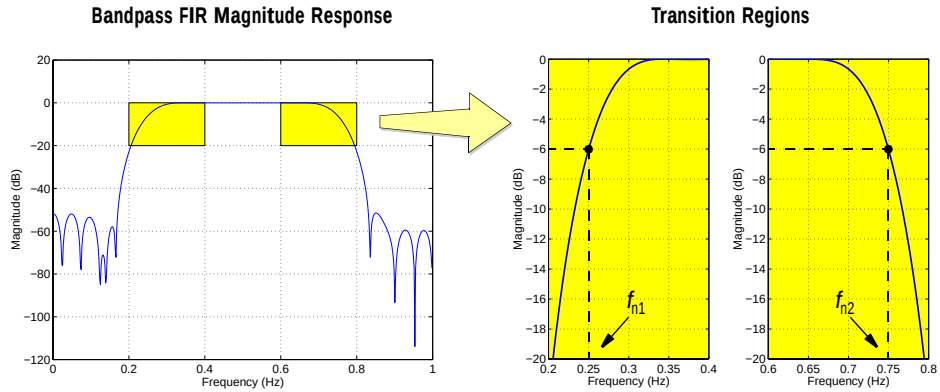
- **Filter type** = **Lowpass**
- **Order** = `41`
- **Cutoff frequency** = `0.5`

The cutoff frequency for a highpass filter (**Highpass** selected from the **Filter type** menu) is defined in the same way: -6 dB down from the passband level. The equivalent MATLAB code for the highpass filter design is

```
b = fir1(41,0.5,'high')
```

For highpass designs, the filter order must be even (so that the filter is Type I). MATLAB therefore automatically increases the filter order to 42 for this example.

Bandpass and Bandstop Discrete-Time FIR Filters. The normalized cutoff frequencies for a bandpass filter are illustrated in the next figure. The normalized lower cutoff frequency, f_{n1}, is 0.25, and the normalized upper cutoff frequency, f_{n2}, is 0.75. For an input signal sampled at 1 kHz, these would be equivalent to cutoff frequencies of 150 Hz and 375 Hz, respectively.



The equivalent MATLAB code for this filter design is

```
b = fir1(41,[0.25 0.75])
```

and the filter can be designed in the Digital FIR Filter Design block by making the following selections in the dialog box:

- **Filter type** = **Bandpass**
- **Order** = `41`
- **Lower cutoff frequency** = `0.25`
- **Upper cutoff frequency** = `0.75`

The lower and upper cutoff frequencies for a bandstop filter (**Bandstop** selected from the **Filter type** menu) are defined in the same way: -6 dB down from the passband level. The equivalent MATLAB code for this filter design is

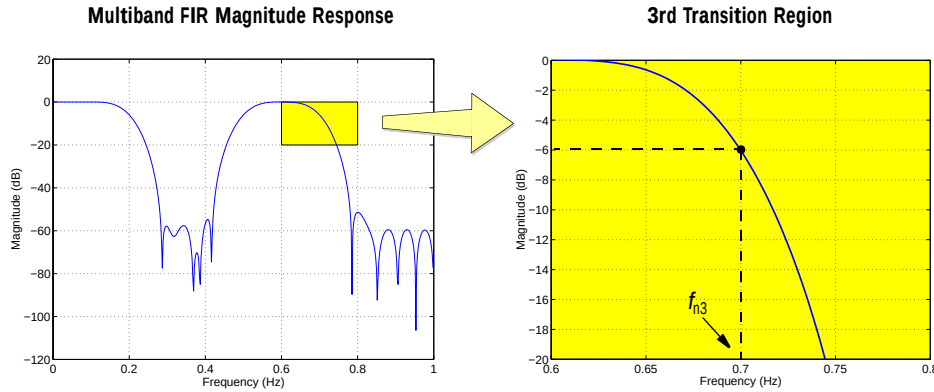
```
b = fir1(41,[0.25 0.75],'stop')
```

Whenever the last (right-most) band is a passband, the filter order must be even (so that the filter is Type I). MATLAB therefore automatically increases the filter order to 42 for the stopband design.

Multiband Discrete-Time FIR Filters. The normalized cutoff frequencies for a multiband filter are illustrated in the next figure. Multiband filters are constructed in the same way as bandpass or bandstop filters, except that a *vector* of cutoff frequencies, f_n, is specified (instead of just an upper and lower cutoff frequency pair). In the figure, the normalized cutoff frequencies are

```
fn = [0.2 0.5 0.7]
```

For an input signal sampled at 1 kHz, these would be equivalent to cutoff frequencies of 100 Hz, 250 Hz, and 350 Hz, respectively. The right plot shows a zoomed view of the third transition region, defined by cutoff frequency `fn(3)`.



The equivalent MATLAB code for this filter design is

```
b = fir1(41,[0.2 0.5 0.7],'dc-1')
```

and the filter can be designed in the Digital FIR Filter Design block by making the following selections in the dialog box:

- **Filter type** = **Multiband**

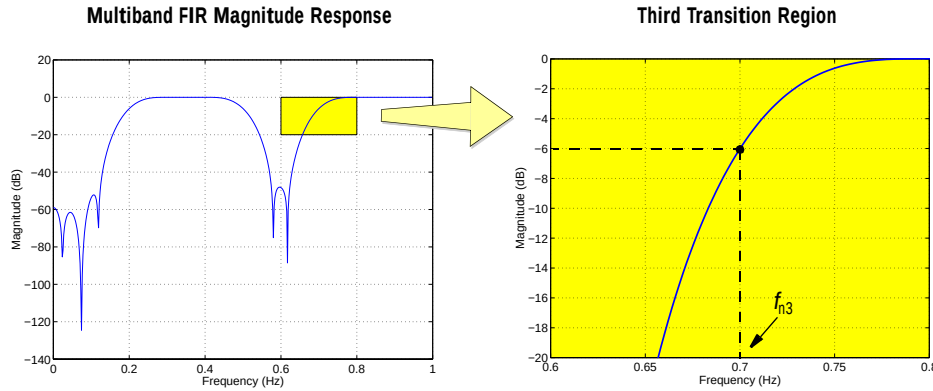
- **Order** = `41`
- **Cutoff frequency vector** = `[0.2 0.5 0.7]`
- **Gain in the first band** = `1`

A multiband filter response alternates between passband (gain ≈ 1) and stopband (gain ≈ 0) for however many bands are specified in the cutoff frequency vector. The **Gain in the first band** indicates whether the band pattern begins with a stopband or passband: `0` indicates an initial stopband, and `1` indicates an initial passband. In general, if the first (leftmost) band is a passband, as shown above, the frequency vector describes the response as follows:

`fn(1)` = first passband-to-stopband cutoff frequency
`fn(2)` = first stopband-to-passband cutoff frequency
`fn(3)` = second passband-to-stopband cutoff frequency
`fn(4)` = second stopband-to-passband cutoff frequency
⋮ ⋮ ⋮ ⋮

If the first (leftmost) band is a stopband, as shown below, the frequency vector describes the response as follows:

`fn(1)` = first stopband-to-passband cutoff frequency
`fn(2)` = first passband-to-stopband cutoff frequency
`fn(3)` = second stopband-to-passband cutoff frequency
`fn(4)` = second passband-to-stopband cutoff frequency
⋮ ⋮ ⋮ ⋮



The equivalent MATLAB code for an initial-stopband response is

```
b = fir1(41,[0.2 0.5 0.7],'dc-0')
```

Whenever the last (right-most) band is a passband, as it is above, the filter order must be even (so that the filter is Type I). MATLAB therefore automatically increases the filter order to 42 for this example.

Arbitrary Shape Discrete-Time FIR Filters. Arbitrary shape filters are the most flexible filter designs and are constructed by specifying a normalized *band-edge frequency* vector (instead of a normalized *cutoff-frequency* vector). The normalized frequency vector, `fn`, contains frequency points in the range `0` to `1` (inclusive) in ascending order. A magnitude vector, `mn`, specifies the desired normalized magnitude response at the corresponding points in the frequency vector.

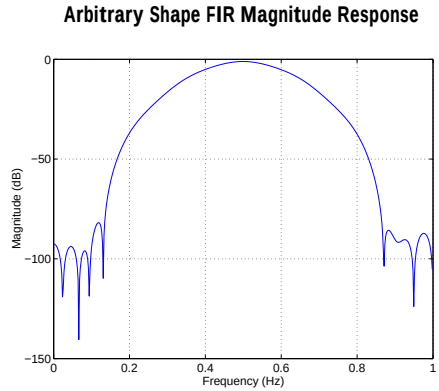
The desired magnitude response of the design can therefore be displayed by typing

```
plot(fn,mn)
```

In the figure below, the normalized band-edge frequencies and magnitudes are

```
fn = [0.0 0.2 0.3 0.5 0.7 0.8 1.0]
mn = [0.0 0.0 0.1 1.0 0.1 0.0 0.0]
```

For an input signal sampled at 1 kHz, this filter would have stopbands from DC to 100 Hz and from 400 Hz to 500 Hz. The response in the passband is specified to have normalized gain of 0.1 (-20 dB) at 150 Hz and 350 Hz, and normalized gain of 1 at 250 Hz.



The equivalent MATLAB code, with corresponding frequency-magnitude pairs shaded, is

```
b = fir2(41,[0.0 0.2 0.3 0.5 0.7 0.8 1.0],...
             [0.0 0.0 0.1 1.0 0.1 0.0 0.0]);
```

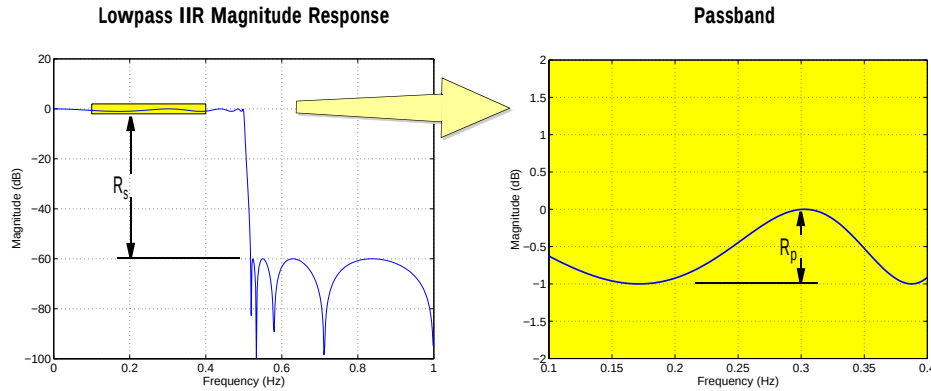
and the filter can be designed in the Digital FIR Filter Design block by making the following selections in the dialog box:

- **Filter type** = **Arbitrary shape**
- **Order** = `41`
- **Frequency vector** = `[0.0 0.2 0.3 0.5 0.7 0.8 1.0]`
- **Gains at these frequencies** = `[0.0 0.0 0.1 1.0 0.1 0.0 0.0]`

Note that normalized frequency-magnitude pairs must be provided for spectrum boundaries, 0 and 1. Whenever the last (right-most) band is a passband, the filter order must be even (so that the filter is Type I). If the example above had a passband at half the sample frequency, MATLAB would automatically increase the filter order to 42.

Discrete-Time IIR Filters. For IIR filters, the cutoff frequency is defined as the frequency where the magnitude response drops to -3 dB below the passband level (one half the passband power). Passband ripple, R_p, is defined as the peak-to-peak ripple magnitude in the passband, and stopband attenuation, R_s, is defined as the peak magnitude in the stopband. The following figure shows the response of an elliptic lowpass IIR filter with 1 dB of passband ripple and

60 dB of stopband attenuation. The right plot shows a zoomed view of the passband.

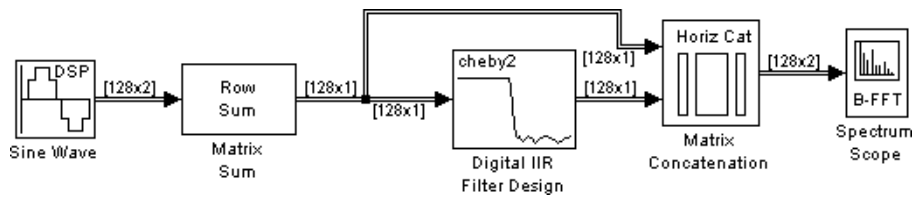


The Digital IIR Filter Design block uses the following Signal Processing Toolbox functions to design its filters:

- 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`.

For more information on the filter design algorithms, see the Signal Processing Toolbox documentation.

Example: Chebyshev Type II Lowpass Filter. In the model below, the Sine Wave block generates two sinusoids, one at 100 Hz and 400 Hz, each with a frame size of 128 samples. The sinusoids are then summed point-by-point, and the resulting signal is passed through a lowpass IIR filter to attenuate the higher frequency sinusoid. Both signals are displayed on the scope.



To build the model, make the following settings:

- In the Sine Wave block, set:
  - **Amplitude** = `10`
  - **Frequency** = `[100 400]`
  - **Phase offset** = `0`
  - **Sample time** = `0.001`
  - **Samples per frame** = `128`
- In the Matrix Sum block, set **Sum along** = **Rows**.
- In the Digital IIR Filter Design block, set:
  - **Design method** = **Chebyshev II**
  - **Filter type** = **Lowpass**
  - **Filter order** = `8`
  - **Stopband edge frequency** = `0.5`
  - **Stopband attenuation in dB** = `20`

  This specifies a Chebyshev type II lowpass filter with a stopband edge frequency of 250 Hz, which will attenuate the sinusoid at 400 Hz but retain the sinusoid at 100 Hz. Note that half the sample frequency in this case is 500 Hz, so that the normalized **Stopband edge frequency** is `0.5`, as specified.
- In the Matrix Concatenation block, set:
  - **Number of input** = `2`
  - **Concatenation method** = **Horizontal**
- Set the **Stop time** in the **Parameters** dialog box to `inf`, and start the simulation by selecting **Start** from the **Simulation** menu.

Use the default setting in the Spectrum Scope block. As the simulation begins running, right-click in the plot area of the scope, and select **Autoscale** from the pop-up menu. You can also change the styles and colors of the plotted lines by selecting either **CH 1** or **CH 2** from the right-click pop-up menu.

The scope window displays both the original signal and the filtered result, which shows the expected attenuation of the 400 Hz component. Stop the simulation at any time by selecting **Stop** from the **Simulation** menu.

Designing Continuous-Time Classical IIR Filters

The Analog Filter Design block designs and applies continuous-time 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 table below.

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

where:

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

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

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

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

For more information on the filter design algorithms, see the Signal Processing Toolbox documentation.

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

Designing Discrete-Time Special IIR and FIR Filters

The Remez FIR Filter Design, Yule-Walker IIR Filter Design, and Least Squares FIR Filter Design blocks design and implement IIR or FIR filters with arbitrary magnitude responses, including multiband responses.

The Yule-Walker IIR Filter Design block designs recursive IIR digital filters by fitting a specified frequency response based on arbitrary piecewise linear magnitude responses. For more on the Yule-Walker algorithm, see the description of the `yulewalk` function in the Signal Processing Toolbox.

The Remez FIR Filter Design and Least Squares FIR Filter Design blocks design FIR filters using the Parks-McClellan and least-squares techniques, respectively. These techniques reflect two error minimization schemes that provide optimal fits to a desired frequency response, each using a different definition of "optimal fit":

- The Remez FIR Filter Design block implements the Parks-McClellan algorithm, which uses the Remez exchange algorithm and Chebyshev approximation theory to design filters with optimal fits between the desired and actual frequency responses. The filters are optimal in the sense that they minimize the maximum error between the desired frequency response and the actual frequency response over the designated bands. Filters designed in this way exhibit an equiripple behavior in their frequency response, and hence are also known as *equiripple* filters.
- The Least Squares FIR Filter Design block minimizes the integral of the squared error between the desired frequency response and the actual frequency response. This technique provides a better response over most of the passband and stopband than does the Parks-McClellan algorithm. At the

band edges, however, the least-squares technique provides a poorer fit than does an equiripple filter designed to fit the same response.

For more on the Parks-McClellan and least squares design techniques, see the descriptions of the `remez` and `firls` functions in the Signal Processing Toolbox.

The following four sections provide more information about the parameters provided by these blocks:

- "Frequency and Magnitude Parameter Overview"
- "Frequency and Magnitude Parameters: Yule-Walker IIR Filter Design"
- "Frequency and Magnitude Parameters: Remez and Least Squares FIR Filter Design"
- "Weight Parameters"

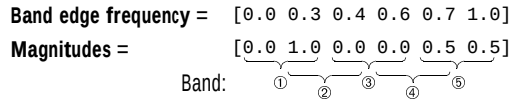
The final section, "Example: Least Squares Multiband Filter" on page 4-21, provides an example that uses the Least Squares FIR Filter Design block.

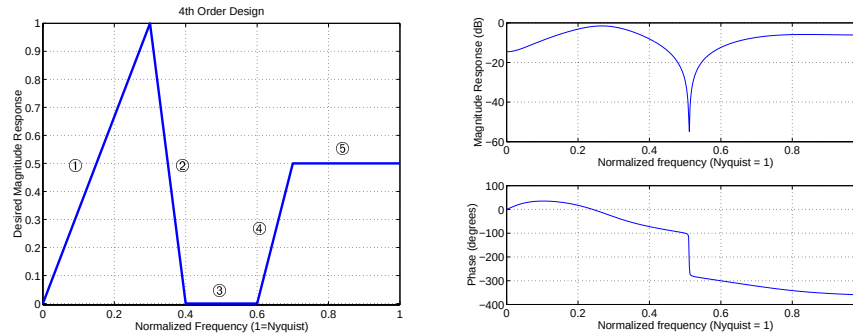
Frequency and Magnitude Parameter Overview. All of the special IIR and FIR blocks let you design filters with arbitrary magnitude responses. The magnitude response can include multiple stopbands, passbands, and transition regions. You specify the desired frequency response using the blocks' **Band edge frequency vector** and **Magnitudes at these frequencies** or **Gains at these frequencies** parameters. These parameters specify the range and magnitude, respectively, of the frequency bands that make up the filter's frequency response.

Frequency and Magnitude Parameters: Yule-Walker IIR Filter Design. For the Yule-Walker IIR Filter Design block, the **Band edge frequency vector** and **Magnitudes at these frequencies** parameters describe a piecewise linear magnitude response over the entire frequency range. The **Band edge frequency vector** contains points in the range `0` to `1`, where `1` corresponds to half the sampling frequency. The **Magnitudes at these frequencies** vector contains the desired magnitude response at the points in the **Band edge frequency vector**. The two vectors must be the same length. The "transition regions" in this case are the linear segments connecting the defined frequency-magnitude pairs.

For example, the simple bandstop frequency response in the figure below can be specified by entering the following parameter values:

- **Band edge frequency vector** = `[0 0.3 0.4 0.6 0.7 1]`, which specifies the desired frequency points.
- **Magnitudes at these frequencies** = `[0 1 0 0 0.5 0.5]`, which defines the magnitudes corresponding to the frequencies above.





Together, the **Band edge frequency vector** and **Magnitudes at these frequencies** parameters shown define:

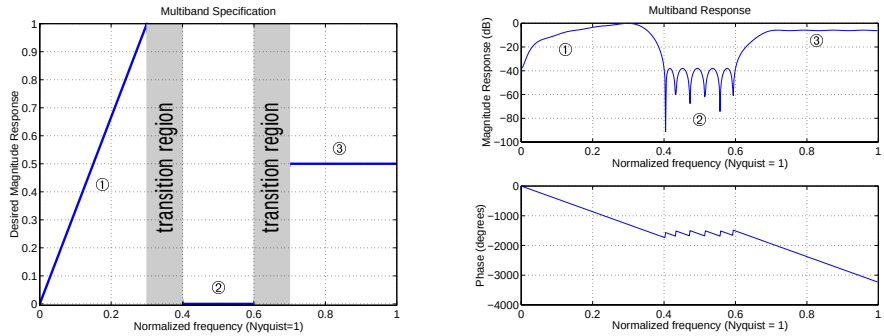
- A stopband, from `0.4` to `0.6`
- A passband from `0.7` to `1`
- Three transition regions: `0` to `0.3`, `0.3` to `0.4`, and `0.6` to `0.7`

Frequency and Magnitude Parameters: Remez and Least Squares FIR Filter Design. For the Remez FIR Filter Design and Least Squares FIR Filter Design blocks, the **Band edge frequency vector** and **Gains at these frequencies** vectors describe linear magnitude *segments*, as shown below. The distances between segments represent "don't care" or transition regions. Both vectors must have even length.

Band edge frequency = [0 0.3 0.4 0.6 0.7 1]

Gains = [0 1 0 0 0.5 0.5]

Band: ① ② ③



Weight Parameters. The Remez FIR Filter Design and Least Squares FIR Filter Design blocks allow you to weight the error minimization in certain frequency bands by entering a vector for the band **Weights**. The **Weights** parameter is useful when designing a compound filter (for example, a lowpass differentiator). For example, to specify a lowpass filter with a transition region in the normalized frequency range 0.4 to 0.5, and 10 times more error minimization in the stopband than the passband, use:

- **Band edge frequency vector** = `[0 0.4 0.5 1]`
- **Magnitudes at these frequencies** = `[1 1 0 0]`
- **Weights** = `[1 10]`

The **Weights** vector is always half the length of the **Band edge frequency vector** and **Magnitudes at these frequencies** vectors; there must be exactly one weight per band.

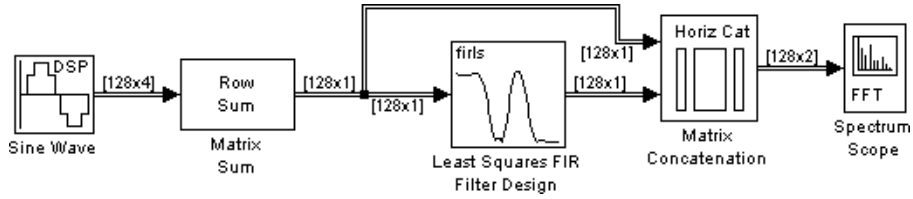
The **Weights** vector is interpreted differently when **Hilbert Transformer** or **Differentiator** is selected from the **Filter type** parameter:

- **Differentiator** –The differentiator designs use special weighting techniques for nonzero magnitude bands. The Remez FIR Filter Design block assumes that the weight is equal to the inverse of the frequency multiplied by the weight specified in the **Weights** vector. The Least Squares FIR Filter Design

block assumes that the weight is equal to the inverse of the frequency squared, multiplied by the weight specified in the **Weights** vector. In each case, the result is a filter with much better fit at low frequencies than at high frequencies. In most cases, however, differentiators have only a single band, so the weight is a scalar value that does not affect the final filter.

- **Hilbert Transform** – The Hilbert transform designs apply a constant weight in each nonzero magnitude band, simply multiplying the error by the specified weight for that band. Similar to the differentiators, Hilbert transformers usually have only a single band, so the weight is a scalar value that does not affect the final filter.

Example: Least Squares Multiband Filter. In the model below, the Sine Wave block generates four sinusoids, at 100, 200, 300, and 400 Hz, each with a frame size of 128 samples. The sinusoids are summed point-by-point, and the resulting signal is passed through a multiband FIR filter to attenuate the sinusoids at 200 and 400 Hz. Both signals are displayed on the scope.



To build the model, make the following parameter settings:

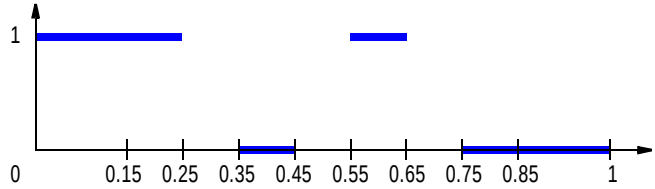
- In the Sine Wave block, set:
  - **Amplitude** = `10`
  - **Frequency** = `[100 200 300 400]`
  - **Phase** = `0`
  - **Sample time** = `0.001`

  The **Sample time** setting represents a sample rate of 1 kHz.
- In the Matrix Sum block, set **Sum along** = **Rows**.
- In the Least Squares FIR Filter Design block, set:
  - **Filter type** = **Multiband**
  - **Filter order** = `16`
  - **Band edge frequency vector** = `[0 125 175 225 275 325 375 500]/500`

  - **Gains at these frequencies** = `[1 1 0 0 1 1 0 0]`
  - **Weights** = `[1 1 1 1]`
- In the Matrix Concatenation block, set:
  - **Number of input** = `2`
  - **Concatenation method** = **Horizontal**
- Set the **Stop time** in the **Parameters** dialog box to `inf`, and start the simulation by selecting **Start** from the **Simulation** menu.

Use the default setting in the Spectrum Scope block. As the simulation begins running, right-click in the plot area of the scope, and select **Autoscale** from the pop-up menu. You can also change the styles and colors of the plotted lines by selecting either **CH 1** or **CH 2** from the right-click pop-up menu.

The **Band edge frequency vector** and **Gains at these frequencies** vectors define the desired magnitude response below.



Note that the values in the **Band edge frequency vector** are divided by half the sample frequency (in this case is 500 Hz) so that the maximum value is `1`, as required.

The scope window displays both the original signal and the filtered result, which shows the expected attenuation of the 200 and 400 Hz components. Stop the simulation at any time by selecting **Stop** from the **Simulation** menu. Try different band configurations to attenuate the peaks at different frequencies.

Filter Structures

Any realizable filter can be represented in the time domain by a difference equation of the form

$$y(k) = b_1 u(k) + b_2 u(k-1) + \cdots + b_m u(k-m-1) - a_2 y(k-1) - \cdots - a_n y(k-n-1)$$

where $y(k)$ and $u(k)$ are, respectively, the output and input at the current time step, $y(k\text{-}1)$ and $u(k\text{-}1)$ are the output and input at the previous time step, and so on. The values b_1, b_2, ..., b_m, and a_2, ..., a_n are the filter coefficients, or *taps*.

Every realizable filter is therefore fundamentally a collection of multiplications, additions, and delays. The order in which these assorted operations are implemented in practice determines the *filter structure* (also known as the filter *realization*, *architecture*, or *implementation*). Implementations may differ from each other in terms of speed, memory requirements, delay, and quantization error. See “Linear System Models” in the Signal Processing Toolbox documentation for more information about common filter structures.

The Filter Structures library provides a number of blocks for filter implementation:

- Analog Filter Design
- Biquadratic Filter
- Direct-Form II Transpose Filter
- Overlap-Add FFT Filter
- Overlap-Save FFT Filter
- Time-Varying Direct-Form II Transpose Filter
- Time-Varying Lattice Filter

Additionally, the library includes the Filter Realization Wizard for creating a variety of custom designs.

The DSP Blockset provides a number demos that make use of the filter structure blocks:

- Frequency Domain Filtering (`olapfilt`)
- LPC Analysis and Synthesis of Speech (`dsplpc`)
- Sample Rate Conversion (`dspsrcnv`)

Multirate Filters

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

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

- Dyadic Analysis Filter Bank
- Dyadic Synthesis Filter Bank
- FIR Decimation
- FIR Interpolation
- FIR Rate Conversion
- Wavelet Analysis
- Wavelet Synthesis

The DSP Blockset additionally provides a number of demos that make use of the multirate filter blocks:

- Denoising (`dspwdnois`)
- Multistage Multirate Filtering Suite (`dspmrf_menu`)
- Interpolation of a Sinusoidal Signal (`dspintrp`)
- Sample Rate Conversion (`dspsrcnv`)
- Sigma-Delta A/D Converter (`dspsdadc`)
- Three-Channel Wavelet Transmultiplexer (`dspwvtrnsmx`)
- Wavelet Perfect Reconstruction Filter Bank (`dspwpr1`)
- Wavelet Reconstruction (`dspwlet`)

Transforms

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

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

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

Using the FFT and IFFT Blocks

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

- "Example: Using the FFT Block"
- "Example: Using the IFFT Block"

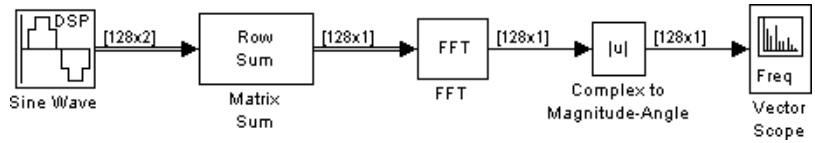
The first example loosely follows the example in the "Discrete Fourier Transform" section of the Signal Processing Toolbox documentation, where you can also find additional background information on these transform operations.

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

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

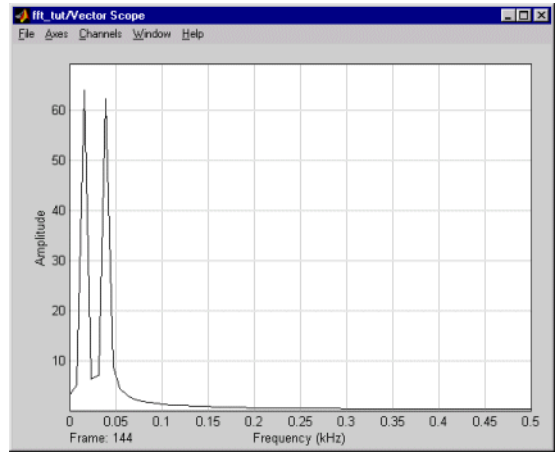
which is then transformed to the frequency domain using an FFT block.



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.

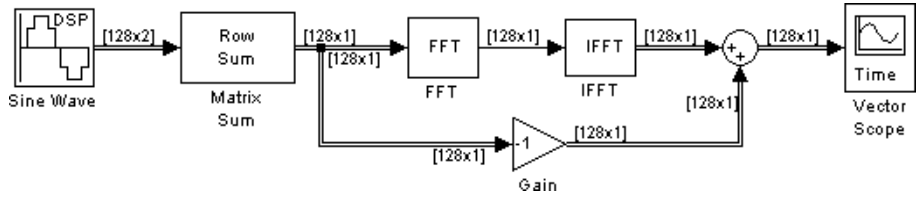
Other blocks that compute the FFT internally are the blocks in the Power Spectrum Estimation library. See “Power Spectrum Estimation” on page 4-30 for more information about these blocks.

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.

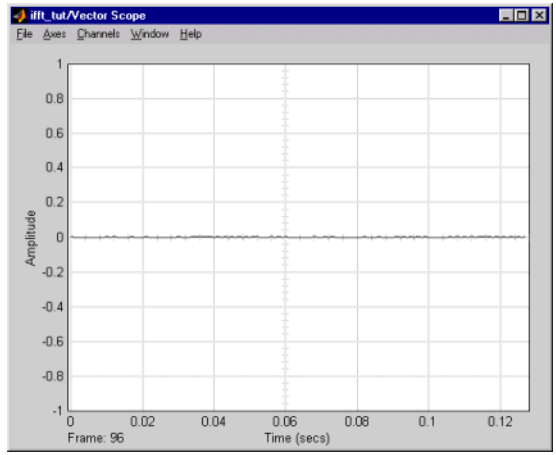


To build the model, make the following parameter settings:

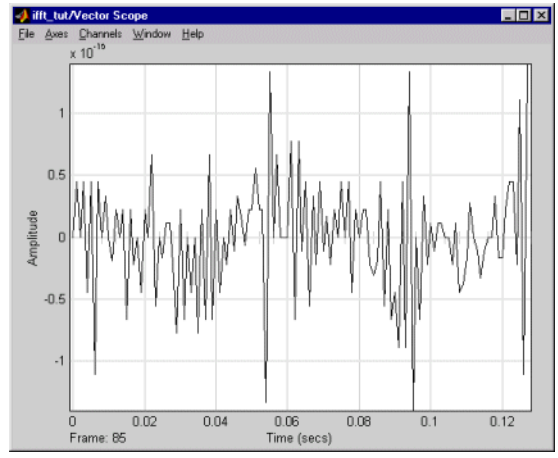
- In the Sine Wave block, set:
  - **Amplitude** = `1`
  - **Frequency** = `[15 40]`
  - **Phase offset** = `0`
  - **Sample time** = `0.001`
  - **Samples per frame** = `128`
- In the Matrix Sum block, set **Sum along** = **Rows**.
- In the Sum block, set **List of signs** = `|++`.
- In the Gain block, 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}.


ifft_tut/Vector Scope
File Axes Channels Window Help
x 10^-19
1
0.5
0
-0.5
-1
Amplitude
0
0.02
0.04
0.06
0.08
0.1
0.12
Frame: 85
Time (secs)

Power Spectrum Estimation

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

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

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

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

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

Linear Algebra

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

- Linear System Solvers
- Matrix Factorizations
- Matrix Inverses

A third library, Matrix Operations, provides other essential blocks for working with matrices. See “Multichannel Signals” on page 3-11 for more information about matrix signals.

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

- “Solving Linear Systems”
- “Factoring Matrices”
- “Inverting Matrices”

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

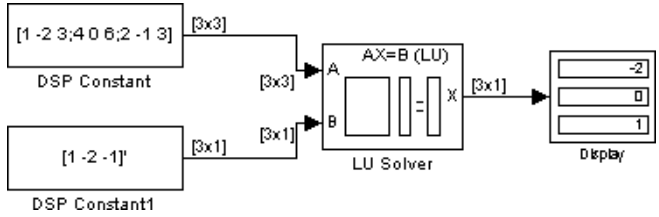
Some of the blocks offer particular strengths for certain classes of problems. For example, the Cholesky Solver block is particularly adapted for a square Hermitian positive definite matrix A, whereas the Backward Substitution block is particularly suited for an upper triangular matrix A.

Example: LU Solver

In the model below, the LU Solver block solves the equation A*x* = b, where

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

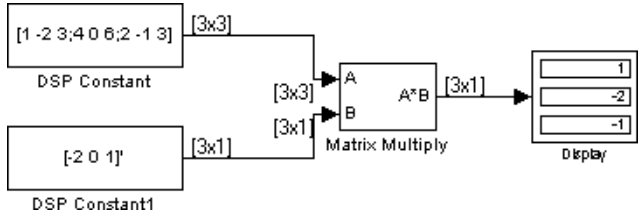
and finds *x* to be the vector `[-2 0 1]'`.



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 A*x*, 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

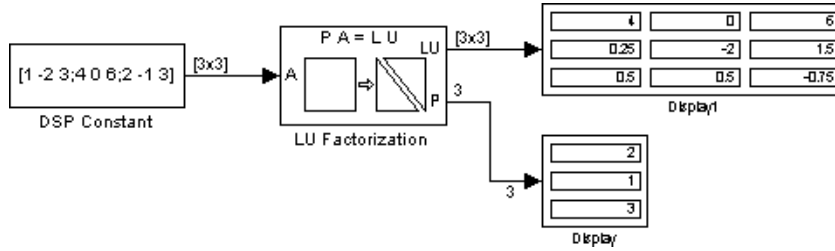
- QR Factorization
- Singular Value Decomposition

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]`.

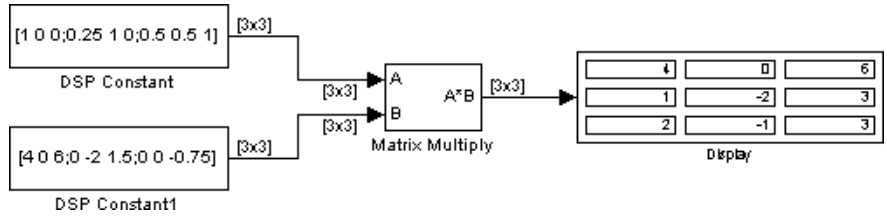
The lower output of the LU Factorization, `P`, is the permutation index vector, which indicates that the factored matrix A_p is generated from A by interchanging the first and second rows.

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

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:

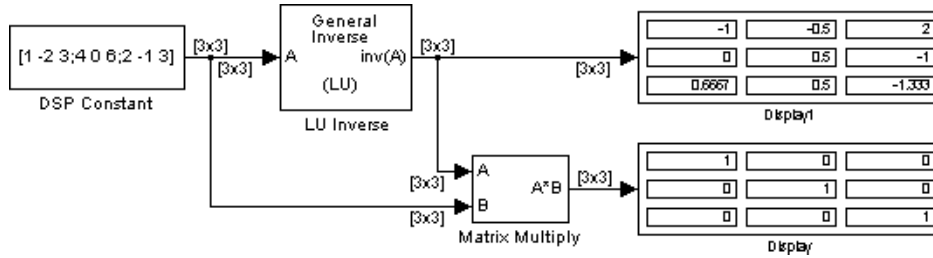
- Cholesky Inverse
- LDL Inverse
- LU Inverse
- Pseudoinverse

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

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

Statistics

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

- Basic operations
- Running operations

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

- Histogram
- Mean
- RMS
- Standard Deviation
- Variance

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

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

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

- "Basic Operations"
- "Running Operations"

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

Basic Operations

A *basic operation* is one that processes each input independently of previous and subsequent inputs. For example, in basic mode (with **Value and Index**

selected, for example) the Maximum block finds the maximum value in each column of the current input, and returns this result at the top output (`Val`). Each consecutive `Val` output therefore has the same number of columns as the input, but only one row. Furthermore, the values in a given output only depend on the values in the corresponding input. The block repeats this operation for each successive input.

This type of operation is exactly equivalent to the MATLAB command

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

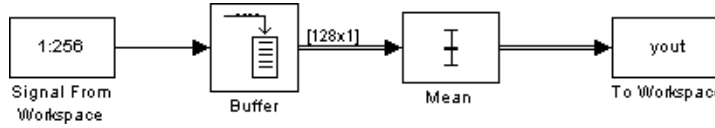
which computes the maximum of each column in input `u`.

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

Example: Sliding Windows

You can use the basic statistics operations in conjunction with the Buffer block to implement basic sliding window statistics operations. A *sliding window* is like a stencil that you move along a data stream, exposing only a set number of data points at one time.

For example, you may want to process data in 128-sample frames, moving the window along by one sample point for each operation. One way to implement such a sliding window is shown in the model below.



The Buffer block's **Buffer size** (M_0) parameter determines the size of the window. The **Buffer overlap** (L) parameter defines the "slide factor" for the window. At each sample instant, the window slides by M_0-L points. The **Buffer overlap** is often M_0-1 (the same as the Delay Line block), so that a new statistic is computed for every new signal sample.

To build the model, make the following settings:

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

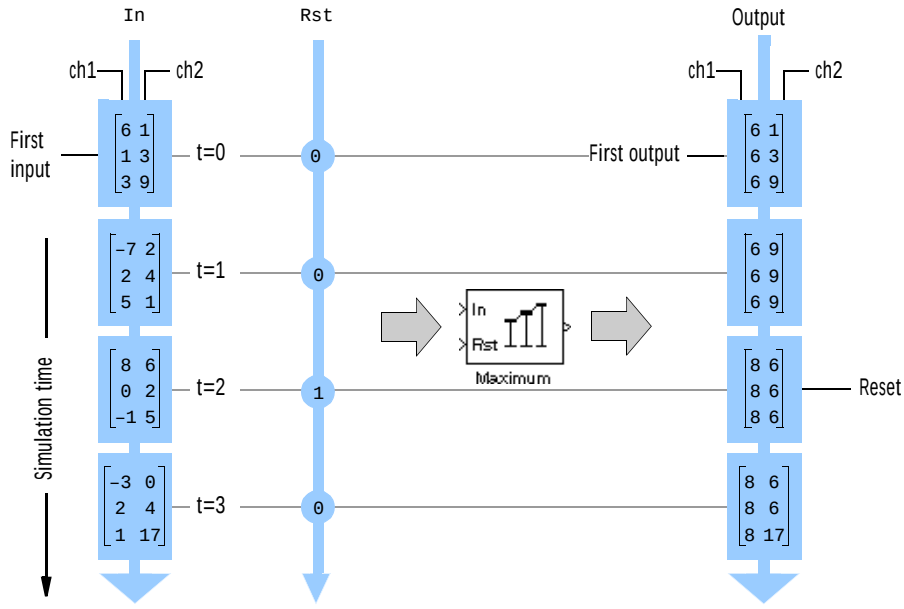
- In the Buffer block, set:
  - **Output buffer size (per channel)** = 128
  - **Buffer overlap** = 127

Running Operations

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

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

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



DSP Blockset Demos Overview

You can access the DSP Blockset demos by typing

```
demos
```

at the MATLAB command line. In the Demos window that opens, expand the **Blocksets** entry by double-clicking, and then click **DSP** to see the demos.

Explore all the demos to see how you can implement both basic and advanced DSP algorithms with the DSP Blockset. You can also use the demos as a base for building your own models. Simply select the section of the demo that you want to build on and copy it into your own model.

The available demos are listed below by category.

Adaptive Processing Demos

- Equalization: Demonstrates adaptive channel equalization by using the LMS algorithm to adaptively compute an estimate of an FIR equalization filter.
- Noise canceller (using either LMS or RLS): These demos use either the LMS or RLS algorithm to subtract noise from an input signal.
- Linear prediction: Uses the LMS adaptive FIR algorithm to adaptively compute the linear prediction coefficients for a noisy input signal.
- Time-delay estimation: Uses the LMS adaptive FIR algorithm to adaptively estimate the time delay for a noisy input signal.
- Tracking filter: Uses a Kalman filter to track the time-varying weights of a nonstationary fifth order FIR filter.

Audio Processing Demos

- Dynamic range compression: Compresses the dynamic range of a signal by modifying the range of the magnitude at each frequency bin. This nonlinear spectral modification is followed by an overlap-add FFT algorithm for reconstruction.
- Flanging: Introduces a “flanging” effect into a short segment of music.
- Reverberation: Uses the Integer Delay block to demonstrate the popular reverberation audio effect.

- LPC analysis and synthesis: Uses the Levinson solver and Time-Varying Lattice Filter for low-bandwidth transmission of speech.
- Waveform coding: This set of demos uses a variety of modulation methods to code a waveform using one bit per message sample:
  - ADPCM (Adaptive Differential Pulse Code Modulation)
  - CVSD (Continuously Variable Slope Delta-modulation)
  - LDM (Linear Delta Modulation)
  - Comparison of LDM, CVSD, and ADPCM

Communications Demos

- SSB modulation: Demonstrates single sideband (SSB) modulation in sample-based and frame-based modes.
- WWV digital receiver: WWV is the call sign of a US Government radio station that transmits frequency reference standards and time code information with a timing accuracy of 10 microseconds and a frequency accuracy of 1 part in 100 billion. This demo simulates the transmission of a WWV signal and demonstrates implementation of the subsequent receiver and decoder blocks. The receiver design serves as a simple example of the use of Simulink, DSP Blockset, Stateflow® and Real-Time Workshop.

Filtering Demos

- Multirate filtering suite: Uses FIR decimation blocks in multiple stages to filter with very short bandwidths and low computational loads.
- FIR interpolation: Uses the FIR Interpolation block to demonstrate interpolation of a delayed sine wave signal.
- Overlap add/save: Demonstrates filtering of a sinusoid using the Overlap-Add and Overlap-Save FFT blocks.
- Sample rate conversion: Illustrates the efficiency of the FIR rate conversion block by comparing the block with the equivalent process of separate upsampling, FIR filtering, and downsampling.

Queues Demo

- Demo uses a Queue block with a system of selection switches to illustrate pushing and popping elements from a queue.

Sigma-Delta A/D Conversion Demo

- Demo illustrates analog-to-digital conversion using a sigma-delta algorithm implementation.

Sine Wave Generation Demo

- Demo compares different sine wave generation systems.

Spectral Analysis Demo

- Short-time FFT: Uses the Short-Time FFT block to compute and display a spectrogram.
- Comparison of techniques: Uses the Vector Scope block to simultaneously display spectral estimates computed by the Short-Time FFT, Burg Method, and Modified Covariance Method blocks.

Statistical Functions Demo

- Demo illustrates the behavior of several running-statistics blocks that are periodically reset every 100 input samples.

Wavelets Demos

- One-level PR filter bank: Uses the Dyadic Analysis and Dyadic Synthesis blocks to implement a perfect reconstruction filter bank.
- Wavelet function: Uses a sequence of FIR interpolation blocks to reconstruct a wavelet function from filter coefficients.
- Denoising: Uses Analysis and Synthesis blocks to remove noise from an input signal.
- Wavelet transmultiplexer (WTM): Illustrates the perfect reconstruction property of the discrete wavelet transform (DWT) by using a WTM to reconstruct three independent combined signals transmitted over a single communications link.

5

DSP Block Reference

Using the DSP Block Reference Chapter

This chapter contains complete information on every block in the DSP Blockset in a structured, accessible format. You should turn to this chapter when you need to find detailed information on a particular block. There are several ways to access these reference pages online:

- Press the **Help** button in a block dialog box.
- Right-click a block in a model, and select **Help** from the pop-up menu.
- Right-click a block in the Simulink Library Browser, and select **Help** from the pop-up menu.
- Type `doc('block name')` at the MATLAB command line.
- Expand the **DSP Blockset** entry in the Help Navigator, and select **DSP Block Reference**.

To learn the basic concepts behind building DSP models with Simulink, see Chapter 2, “Simulink and the DSP Blockset.” To find out about using blocks together for common DSP tasks, see Chapter 3, “Working with Signals.”

What Each Block Reference Page Contains

The block reference entries appear in alphabetical order and each contains most of the following sections.

Section	Description
“Purpose”	The purpose of the block.
“Library”	The library or libraries where the block can be found.
“Description”	A description of the block’s use.
“Dialog Box”	The block’s dialog box and parameters. Tunable parameters are labeled “Tunable”. See “About Tunable Parameters” below.
“See Also”	A list of related blocks and functions.

About Tunable Parameters

Block dialog box parameters that can be adjusted while a simulation is running are called *tunable* parameters. In the “Dialog Box” section of the block reference pages, these parameters are indicated by the word “Tunable” in the parameter description. Parameters that are not labeled this way are not tunable; changing a nontunable parameter while the simulation is running generates an error, and suspends the simulation until the error dialog box is dismissed.

Block Library List

This section contains the following two subsections:

- "Block Library Hierarchy" – a structured list of the DSP Blockset libraries
- "Block Library Contents" – a listing of all DSP blocks, arranged by library

See "Alphabetical List of Blocks" on page 5-13 for an alphabetical listing of blocks. The pages that follow that section contain reference information for all blocks in the DSP Blockset, arranged in alphabetical order by block name.

Block Library Hierarchy

The DSP Blockset contains the following libraries and sublibraries:

- DSP Sinks
- DSP Sources
- Estimation
  - Estimation: Linear Prediction
  - Estimation: Parametric Estimation
  - Estimation: Power Spectrum Estimation
- Filtering
  - Filtering: Adaptive Filters
  - Filtering: Filter Designs
  - Filtering: Filter Structures
  - Filtering: Multirate Filters
- Math Functions
  - Matrices and Linear Algebra
    - Linear System Solvers
    - Matrix Factorizations
    - Matrix Inverses
    - Matrix Operations
  - Math Functions: Math Operations
  - Math Functions: Polynomial Functions
- Quantizers

- Signal Management
  - Signal Management: Buffers
  - Signal Management: Indexing
  - Signal Management: Signal Attributes
  - Signal Management: Switches and Counters
- Signal Operations
- Statistics
- Transforms

Use the Simulink Library Browser to access the blockset directly through the above hierarchical library list.

Block Library Contents

The DSP blocks in each of these libraries are listed below. Simulink blocks that appear in DSP Blockset libraries (such as Display) are followed by the phrase “Simulink block” in parentheses.

DSP Sinks	
Display (Simulink block)	Time Scope (Simulink Block)
Matrix Viewer	To Wave Device
Signal To Workspace	To Wave File
Spectrum Scope	Triggered To Workspace
Vector Scope	

DSP Sources	
Chirp	Identity Matrix
Constant Diagonal Matrix	Multiphase Clock
Constant Ramp	N-Sample Enable

DSP Sources (Continued)	
Counter	Random Source
DSP Constant	Signal From Workspace
Discrete Impulse	Sine Wave
From Wave Device	Triggered Signal From Workspace
From Wave File	Window Function

Estimation: Linear Prediction	
Autocorrelation LPC	

Estimation: Parametric Estimation	
Burg AR Estimator	Modified Covariance AR Estimator
Covariance AR Estimator	Yule-Walker AR Estimator

Estimation: Power Spectrum Estimation	
Burg Method	Modified Covariance Method
Covariance Method	Short-Time FFT
Magnitude FFT	Yule-Walker Method

Filtering: Adaptive Filters	
Kalman Adaptive Filter	RLS Adaptive Filter
LMS Adaptive Filter	

Filtering: Filter Designs	
Analog Filter Design	Least Squares FIR Filter Design
Digital FIR Filter Design	Remez FIR Filter Design
Digital FIR Raised Cosine Filter Design	Yule-Walker IIR Filter Design
Digital IIR Filter Design	

Filtering: Filter Structures	
Analog Filter Design	Overlap-Add FFT Filter
Biquadratic Filter	Overlap-Save FFT Filter
Direct-Form II Transpose Filter	Time-Varying Direct-Form II Transpose Filter
Filter Realization Wizard	Time-Varying Lattice Filter

Filtering: Multirate Filters	
Dyadic Analysis Filter Bank	FIR Rate Conversion
Dyadic Synthesis Filter Bank	Wavelet Analysis
FIR Decimation	Wavelet Synthesis
FIR Interpolation	

Math Functions: Matrices and Linear Algebra Linear System Solvers	
Autocorrelation LPC	Levinson-Durbin
Cholesky Solver	LU Solver
Forward Substitution	QR Solver
LDL Solver	SVD Solver

Math Functions: Matrices and Linear Algebra Matrix Factorizations	
Cholesky Factorization	QR Factorization
LDL Factorization	Singular Value Decomposition
LU Factorization	

Math Functions: Matrices and Linear Algebra Matrix Inverses	
Cholesky Inverse	LU Inverse
LDL Inverse	Pseudoinverse

Math Functions: Matrices and Linear Algebra Matrix Operations	
Constant Diagonal Matrix	Matrix Scaling
Create Diagonal Matrix	Matrix Square
Extract Diagonal	Matrix Sum
Extract Triangular Matrix	Permute Matrix
Identity Matrix	Reciprocal Condition
Matrix Concatenation (Simulink block)	Submatrix
Matrix 1-Norm	Toeplitz
Matrix Multiply	Transpose
Matrix Product	

Math Functions: Math Operations	
Complex Exponential	dB Gain
Cumulative Sum	Normalization
dB Conversion	Difference

Math Functions: Polynomial Functions	
Least Squares Polynomial Fit	Polynomial Stability Test
Polynomial Evaluation	

Quantizers	
Quantizer (Simulink block)	Uniform Encoder
Uniform Decoder	

Signal Management: Buffers	
Buffer	Stack
Delay Line	Triggered Delay Line
Queue	Unbuffer

Signal Management: Indexing	
Flip	Submatrix
Multiport Selector	Variable Selector
Selector (Simulink block)	

Signal Management: Signal Attributes	
Check Signal Attributes	Convert 2-D to 1-D
Contiguous Copy	Frame Status Conversion
Convert 1-D to 2-D	Inherit Complexity

Signal Management: Switches and Counters	
Counter	Multiphase Clock
Edge Detector	N-Sample Enable
Event-Count Comparator	N-Sample Switch

Signal Operations	
Convolution	Unwrap
Downsample	Upsample
Integer Delay	Variable Fractional Delay
Pad	Variable Integer Delay
Repeat	Window Function
Sample and Hold	Zero Pad

Statistics	
Autocorrelation	Median
Correlation	Minimum
Detrend	RMS
Histogram	Sort
Maximum	Standard Deviation
Mean	Variance

Transforms	
Analytic Signal	IDCT
Complex Cepstrum	IFFT
DCT	Real Cepstrum
FFT	

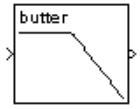
Purpose

Design and implement an analog filter.

Library

Filtering / Filter Designs, Filtering / Filter Structures

Description



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

The input must be a sample-based scalar signal.

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

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

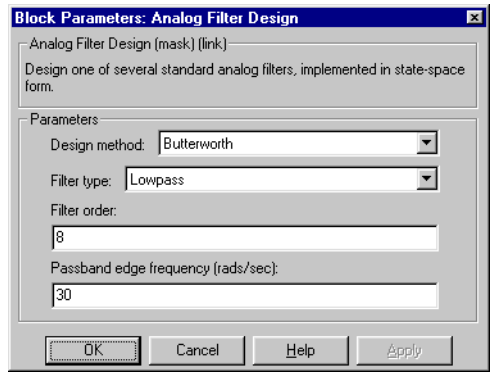
The table below lists the available parameters for each design/band combination. For lowpass and highpass band configurations, these parameters include the passband edge frequency Ω_p, the stopband edge frequency Ω_s, the passband ripple R_p, and the stopband attenuation R_s. For bandpass and bandstop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

attenuation R_s. Frequency values are in rad/s, and ripple and attenuation values are in dB.

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

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

Dialog Box



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

Design method

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

Filter type

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

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.

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.

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.

Stopband edge frequency

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

Lower stopband edge frequency

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

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.

Passband ripple in dB

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

Stopband attenuation in dB

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

References

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

See Also

Digital FIR Filter Design	DSP Blockset
Digital IIR 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:

- "Filter Designs" on page 4-3
- "Filter Structures" on page 4-23

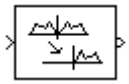
Purpose

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

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

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

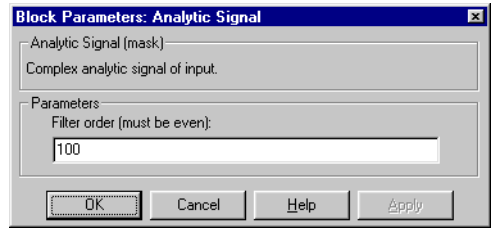
Sample-Based Operation

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

Frame-Based Operation

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

Dialog Box



Filter order

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

See Also Remez FIR Filter Design DSP Blockset

Purpose

Compute the autocorrelation of a vector input.

Library

Statistics

Description

The Autocorrelation block computes the autocorrelation of each column (channel) in an M-by-N input matrix u. Matrix inputs must be frame-based. The result, y, is a frame-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 **All positive lags** is selected, l=M. Otherwise, l is specified as a nonnegative integer by the **Maximum positive lag** parameter.

Input u is zero when indexed outside of its valid range. When the input is real, the output is real; otherwise, the output is complex. If the input is a sample-based vector (row, column, or 1-D), the output is sample-based, with the same shape as the input and length l+1. The Autocorrelation block does not accept a sample-based full-dimension matrix input.

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

- **None** – Generates the raw autocorrelation, $y_{i,j}$, without normalization.
- **Biased** – Generates the biased estimate of the autocorrelation.

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

- **Unbiased** – Generates the unbiased estimate of the autocorrelation.

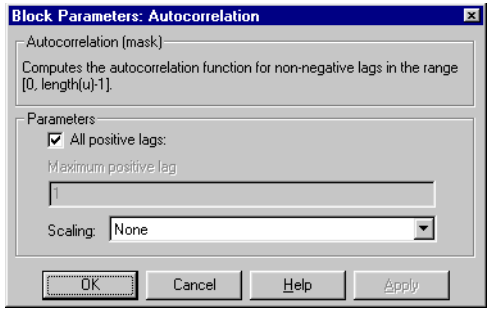
  $$y_{i,j}^{unbiased} = \frac{y_{i,j}}{M-i}$$

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

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

Autocorrelation

Dialog Box



All positive lags

When selected, computes the autocorrelation over all M+1 positive lags.

Maximum positive lag

The maximum positive lag, *l*, for the autocorrelation. This parameter is enabled when the **All positive lags** check box is unselected.

Scaling

The type of scaling for the autocorrelation: **None**, **Biased**, **Unbiased**, or **Unity at zero-lag**. Tunable, except in Simulink's external mode.

See Also

Correlation
`xcorr`
DSP Blockset
Signal Processing Toolbox

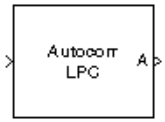
Purpose

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. The input 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.

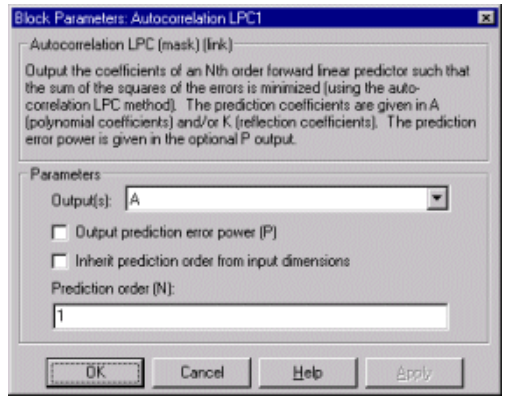
When **Output(s)** is set to **A**, port A is enabled. Port A outputs a length-N+1 column vector whose elements are the prediction error polynomial coefficients. When **Output(s)** is set to **K**, port K is enabled. Port K outputs a length-N column vector whose elements are the prediction error reflection 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 sample-based.

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

See the documentation on the `lpc` function in the Signal Processing Toolbox for details on the algorithms used by the Autocorrelation LPC block.

Autocorrelation LPC

Dialog Box



Outputs

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.

See Also

Levinson-Durbin	DSP Blockset
Yule-Walker Method	DSP Blockset
`lpc`	Signal Processing Toolbox

Backward Substitution

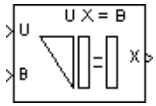
Purpose

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

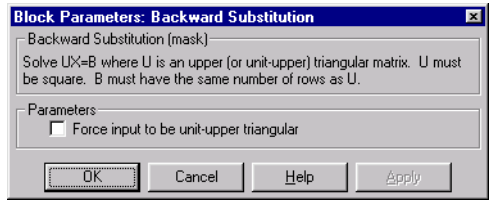


The Backward Substitution block solves the linear system U*X*=B by simple backward substitution of variables, where U is the upper triangular M-by-M matrix input to the `U` port, and B is the M-by-N matrix input to the `B` port. The output is the solution of the equations, the M-by-N matrix *X*, and is always sample-based.

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

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

Dialog Box



Force input to be unit-upper triangular

Replaces the elements on the diagonal of U with 1s when selected. Tunable.

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 4-31 for related information.

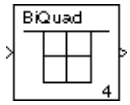
Purpose

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

Library

Filtering / Filter Structures

Description

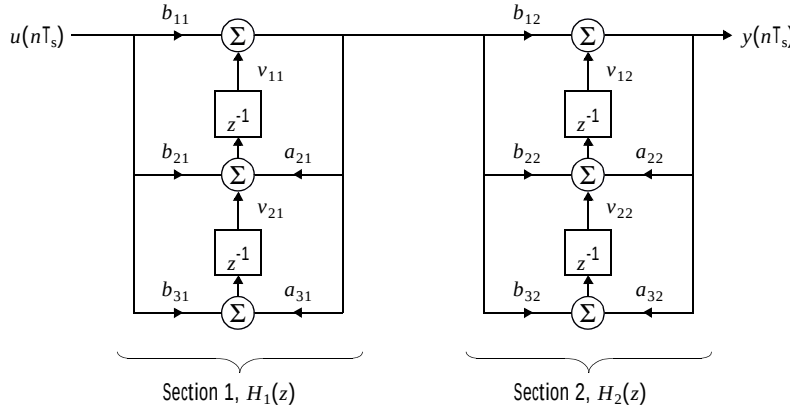


The Biquadratic Filter block applies a cascade of biquadratic filters independently to each input channel. Biquadratic filters are useful for reduced precision implementations because the coefficients are bounded between ±2 for typical minimum-phase designs. This may reduce scaling and coefficient sensitivity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

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

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

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



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

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

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

This is an L-by-6 matrix whose rows contain the numerator and denominator coefficients b_{ik} and a_{ik} of each second-order section in $H(z)$. Use the `ss2sos` and `tf2sos` functions to convert a state-space or transfer-function description of the filter into the second-order section description used by this block. Note that the filter uses a value of 1 for the zero-delay denominator coefficients (a_{11} to a_{1L}) regardless of the value specified in the **SOS matrix** parameter.

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

- *Scalar* to be used for all filter states (v_{11}, v_{12}, ..., v_{1L}, v_{21} v_{22}, ..., v_{2L}) in all channels. An empty vector, `[]`, is the same as the scalar value `0`.
- *Vector* of length 2∗L (row or column) to initialize the filter states for all channels.

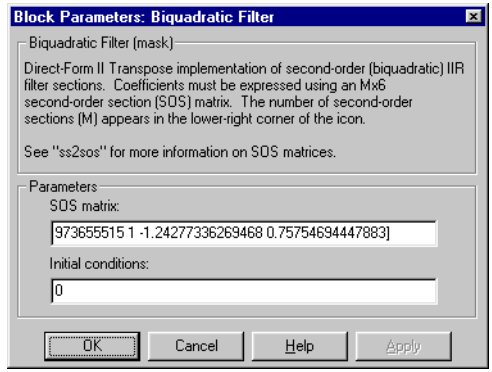
  $$\begin{bmatrix} \underbrace{v_{11}\ v_{21}}_{H_1(z)} & \underbrace{v_{12}\ v_{22}}_{H_2(z)} & \cdots & \underbrace{v_{1L}\ v_{2L}}_{H_L(z)} \end{bmatrix}$$

  Each pair of elements specifies v_{1k} and v_{2k} for second-order section k in every channel.
- *Matrix* of dimension (2∗L)-by-N containing the initial filter states for each of the N channels.

$$
\begin{matrix} H_1(z) \\ \\ H_2(z) \\ \\ \\ H_L(z) \\ \\ \end{matrix}
\left\{
\begin{bmatrix}
v_{11}^{ch1} & v_{11}^{ch2} & v_{11}^{chN} \\
v_{21}^{ch1} & v_{21}^{ch2} & v_{21}^{chN} \\
v_{12}^{ch1} & v_{12}^{ch2} & v_{12}^{chN} \\
v_{22}^{ch1} & v_{22}^{ch2} & v_{22}^{chN} \\
\vdots & \vdots & \vdots \\
v_{1L}^{ch1} & v_{1L}^{ch2} & v_{1L}^{chN} \\
v_{2L}^{ch1} & v_{2L}^{ch2} & v_{2L}^{chN}
\end{bmatrix}
\right.
$$

Each pair of elements in a *column* specifies v_{1k} and v_{2k} for second-order section k of the corresponding channel.

Dialog Box



SOS matrix

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

Initial conditions

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

See Also

Direct-Form II Transpose Filter	DSP Blockset
Filter Realization Wizard	DSP Blockset
Time-Varying Direct-Form II Transpose Filter	DSP Blockset
`filter`	MATLAB
`sosfilt`	Signal Processing Toolbox
`ss2sos`	Signal Processing Toolbox
`tf2sos`	Signal Processing Toolbox

See “Filter Structures” on page 4-23 for related information.

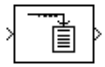
Purpose

Buffer the input sequence to a smaller or larger frame size.

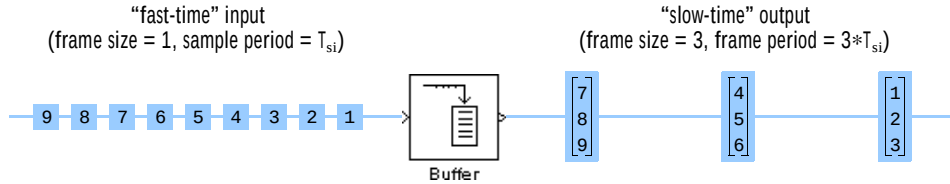
Library

Signal Management / Buffers

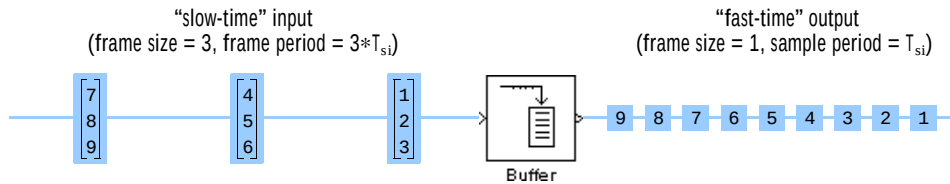
Description



The Buffer block redistributes the input samples to a new frame size, larger or smaller than the input frame size. Buffering to a larger frame size yields an output with a *slower* frame rate than the input, as illustrated below for scalar input.



Buffering to a smaller frame size yields an output with a *faster* frame rate than the input, as illustrated below for scalar output.



The block coordinates the output *frame size* and *frame rate* of nonoverlapping buffers so that the sample period of the signal is the same at both the input and output, $T_{so} = T_{si}$.

Sample-Based Operation

Sample-based inputs are interpreted by the Buffer block as independent channels of data. Thus, a sample-based length-N vector input is interpreted as N independent samples.

In sample-based operation, the Buffer block creates frame-based outputs from sample-based inputs. A sequence of sample-based length-N vector inputs (1-D, 2-D row, or 2-D column) is buffered into an M_o-by-N matrix, where M_o is specified by the **Output buffer size** parameter ($M_o > 1$). That is, each input vector becomes a *row* in the N-channel frame-based output matrix. When

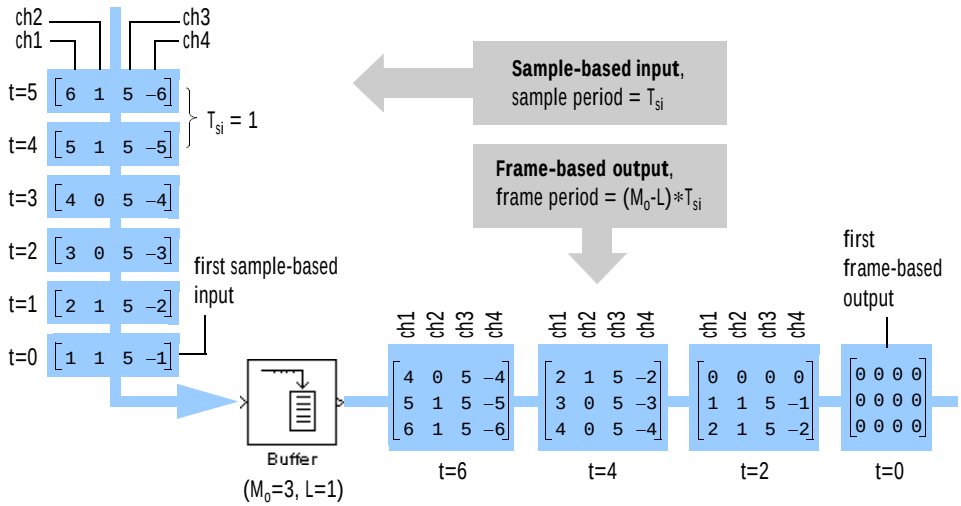
M_o=1, the input is simply passed through to the output, and retains the same dimension.

Sample-based full-dimension matrix inputs are not accepted.

The **Buffer overlap** parameter, L, specifies the number of samples (rows) from the current output to repeat in the next output, where $L < M_o$. For $0 \le L < M_o$, the number of *new* input samples that the block acquires before propagating the buffered data to the output is the difference between the **Output buffer size** and **Buffer overlap**, M_o-L.

The output frame period is $(M_o\text{-}L)*T_{si}$, which is *equal* to the input sequence sample period, T_{si}, when the **Buffer overlap** is M_o-1. For L < 0, the block simply discards L input samples after the buffer fills, and outputs the buffer with period $(M_o\text{-}L)*T_{si}$, which is longer than the zero-overlap case.

In the model below, the block buffers a four-channel sample-based input using a **Output buffer size** of 3 and a **Buffer overlap** of 1.



Note that the input vectors do not begin appearing at the output until the second row of the second matrix. This is due to the block's latency (see "Latency" below). The first output matrix (all zeros in this example) reflects the block's **Initial conditions** setting, while the first row of zeros in the second output is a result of the one-sample overlap between consecutive output frames.

You can use the `rebuffer_delay` function with a frame size of 1 to precisely compute the delay (in samples) for sample-based signals. For the above example,

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

d =
     4
```

This agrees with the four samples of delay (zeros) per channel shown in the figure above.

Frame-Based Operation

In frame-based operation, the Buffer block redistributes the samples in the input frame to an output frame with a new size and rate. A sequence of M_i-by-N matrix inputs is buffered into a sequence of M_o-by-N frame-based matrix outputs, where M_o is the output frame size specified by the **Output buffer size** parameter (i.e., the number of consecutive samples from the input frame to buffer into the output frame). M_o can be greater or less than the input frame size, M_i. Each of the N input channels is buffered independently.

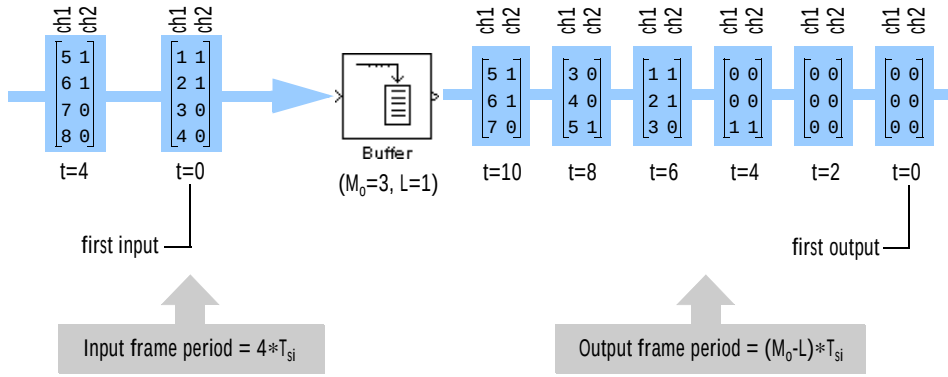
The **Buffer overlap** parameter, L, specifies the number of samples (rows) from the current output to repeat in the next output, where $L < M_o$. For $0 \le L < M_o$, the number of *new* input samples the block acquires before propagating the buffered data to the output is the difference between the **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 Simulink's 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

The Buffer block has *zero tasking latency* in Simulink's single-tasking mode for the following special cases:

- Scalar input and output ($M_o = M_i = 1$) with zero or negative **Buffer overlap** ($L \le 0$)
- Scalar output ($M_o = 1$) with zero **Buffer overlap** ($L = 0$) for any input frame size M_i
- Equal input and output frame sizes ($M_o = M_i$) with zero **Buffer overlap** ($L = 0$)
- Input frame size an integer multiple of the output frame size ($M_i/M_o = k$, for an integer value of k) with zero **Buffer overlap** ($L = 0$)

Zero tasking latency means that the first input sample (received at t=0) appears as first output sample.

Nonzero Latency

Sample-Based Operation. For all cases of *sample-based single-tasking* operation other than those listed above, the Buffer block's buffer is initialized to the value(s) specified by the **Initial conditions** parameter, and the block reads from this buffer to generate the first D output samples, where

$$D = \begin{cases} M_o + L & (L \geq 0) \\ M_o & (L < 0) \end{cases}$$

If the **Buffer overlap**, L, is zero, the **Initial conditions** parameter can be a scalar to be repeated across the first M_o output samples, or a length-M_o vector containing the values of the first M_o output samples. For nonzero **Buffer overlap**, the **Initial conditions** parameter must be a scalar.

Frame-Based Operation. For *frame-based single-tasking* operation and all *multitasking* operation, use the `rebuffer_delay` function to compute the exact delay (in samples) that the Buffer block introduces for a given combination of buffer size and buffer overlap.

For general buffering between arbitrary frame sizes, the **Initial conditions** parameter must be a scalar value, which is then repeated across all elements of the initial output(s). However, in the special case where the *input* is 1-by-N (and the block's output is therefore an M_o-by-N matrix), **Initial conditions** can be:

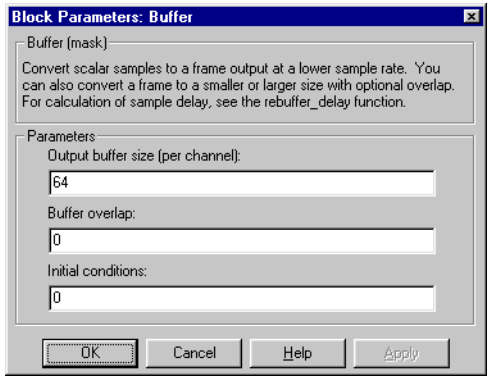
- An M_o-by-N matrix
- A length-M_o vector to be repeated across all columns of the initial output(s)
- A scalar to be repeated across all elements of the initial output(s)

In the special case where the *output* is 1-by-N (the result of unbuffering an M_i-by-N frame-based matrix), **Initial conditions** can be:

- A vector containing M_i samples to output sequentially for each channel during the first M_i sample times
- A scalar to be repeated across all elements of the initial output(s)

See “Excess Algorithmic Delay (Tasking Latency)” on page 3-91 and “The Simulation Parameters Dialog Box” in the Simulink documentation for more information about block rates and Simulink’s tasking modes.

Dialog Box



Output buffer size

The number of consecutive samples, M_o, from each channel to buffer into the output frame.

Buffer overlap

The number of samples, L, by which consecutive output frames overlap.

Initial conditions

The value of the block’s initial output for cases of nonzero latency; a scalar, vector, or matrix.

See Also

Delay Line	DSP Blockset
Unbuffer	DSP Blockset
`rebuffer_delay`	DSP Blockset

See “Buffering Sample-Based and Frame-Based Signals” on page 3-47 for related information.

Purpose

Compute an estimate of AR model parameters using the Burg method.

Library

Estimation / Parametric Estimation

Description

The Burg AR Estimator block uses the Burg method to fit an autoregressive (AR) model to the input data by minimizing (least squares) the forward and backward prediction errors while constraining the AR parameters to satisfy the Levinson-Durbin recursion.

The input is a sample-based vector (row, column, or 1-D) or frame-based vector (column only) representing a frame of consecutive time samples from a single-channel signal, which is assumed to be the output of an AR system driven by white noise. The block computes the normalized estimate of the AR system parameters, $A(z)$, independently for each successive input frame.

$$H(z) = \frac{G}{A(z)} = \frac{G}{1 + a(2)z^{-1} + \ldots + a(p+1)z^{-p}}$$

When **Inherit estimation order from input dimensions** is selected, the order, p, of the all-pole model is one less that the length of the input vector. Otherwise, the order is the value specified by the **Estimation order** parameter

The **Output(s)** parameter allows you to select between two realizations of the AR process:

- **A** – The top output, `A`, is a column vector of length p+1 with the same frame status as the input, and contains the normalized estimate of the AR model polynomial coefficients in descending powers of z,

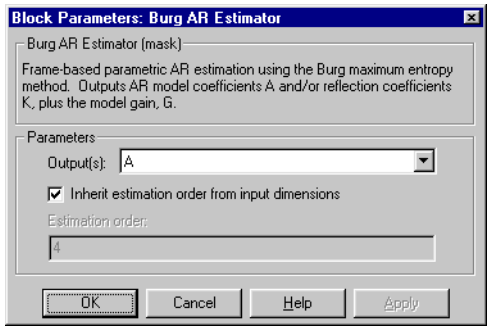
  ```
  [1 a(2) ... a(p+1)]
  ```

- **K** – The top output, `K`, is a column vector of length p with the same frame status as the input, and contains the reflection coefficients (which are a secondary result of the Levinson recursion).
- **A and K** – The block outputs both realizations.

The scalar gain, G, is provided at the bottom output (`G`).

Burg AR Estimator

Dialog Box



Output(s)

The realization to output, model coefficients, reflection coefficients, or both.

Inherit estimation order from input dimensions

When selected, sets the estimation order *p* to one less than the length of the input vector.

Estimation order

The order of the AR model, *p*. This parameter is enabled when **Inherit estimation order from input dimensions** is not selected.

References

Kay, S. M. *Modern Spectral Estimation: Theory and Application*. Englewood Cliffs, NJ: Prentice-Hall, 1988.

Marple, S. L., Jr., *Digital Spectral Analysis with Applications*. Englewood Cliffs, NJ: Prentice-Hall, 1987.

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

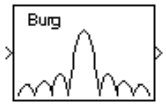
Purpose

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 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.

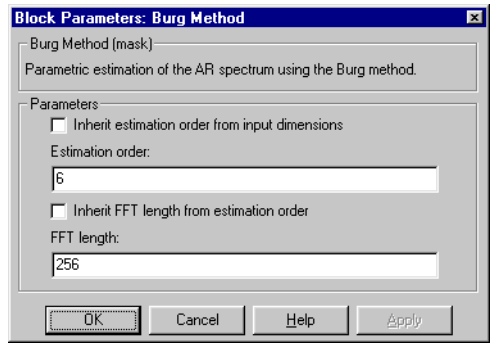
The Burg Method and Yule-Walker Method blocks return similar results for large frame sizes. The following table compares the features of the Burg Method block to the Covariance Method, Modified Covariance Method, and Yule-Walker Method blocks.

	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	
Disadvantages	Peak locations highly dependent on initial phase	May produce unstable models	May produce unstable models	Performs relatively poorly for short data records
	May suffer spectral line-splitting for sinusoids in noise, or when order is very large	Frequency bias for estimates of sinusoids in noise	Peak locations slightly dependent on initial phase	Frequency bias for estimates of sinusoids in noise
	Frequency bias for estimates of sinusoids in noise		Minor frequency bias for estimates of sinusoids in noise	
Conditions for Nonsingularity		Order must be less than or equal to half the input frame size	Order must be less than or equal to 2/3 the input frame size	Because of the biased estimate, the autocorrelation matrix is guaranteed to positive-definite, hence nonsingular

Examples

The dspsacomp demo compares the Burg method with several other spectral estimation methods.

Dialog Box



Inherit estimation order from input dimensions

When selected, sets the estimation order to one less than the length of the input vector.

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 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.

References

Kay, S. M. *Modern Spectral Estimation: Theory and Application*. Englewood Cliffs, NJ: Prentice-Hall, 1988.

Orfanidis, J. S. *Optimum Signal Processing: An Introduction*. 2nd ed. New York, NY: Macmillan, 1985.

Burg Method

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 4-30 for related information.

Purpose

Generate an error when the input signal does or does not match selected attributes exactly.

Library

Signal Management / Signal Attributes

Description

The Check Signal Attributes block terminates the simulation with an error when the input characteristics differ from those specified by the block parameters.

When the **Error if input** parameter is set to **Does not match attributes exactly**, the block generates an error only when the input possesses *none* of the attributes specified by the other parameters. Signals that possess *at least one* of the specified attributes are propagated to the output unaltered, and do not generate an error.

When the **Error if input** parameter is set to **Matches attributes exactly**, the block generates an error only when the input possesses *all* attributes specified by the other parameters. Signals that do not possess *all* of the specified attributes are propagated to the output unaltered, and do not generate an error.

Signal Attributes

The Check Signal Attributes block can test for up to five different signal attributes, as specified by the following parameters. When **Ignore** is selected in any parameter, the block does not check the signal for the corresponding attribute. For example, when **Complexity** is set to **Ignore**, neither real nor complex inputs cause the block to generate an error. The attributes are:

- **Complexity**

  Checks whether the signal is real or complex. (Note that this information can also be displayed in a model by attaching a Probe block with **Probe complex signal** selected, or by selecting **Port data types** from the model window's **Format** menu.)

- **Frame status**

  Checks whether the signal is frame-based or sample-based. (Note that Simulink displays sample-based signals using a single line, →, 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. See"Signal Dimension Nomenclature" on page 1-10 for a description of Simulink signal dimensions. M and N are positive integers unless otherwise indicated below.

Dimensions	Is...	Is not...
1-D	1-D vector, 1-D scalar	M-by-N matrix, 1-by-N matrix (row vector), M-by-1 matrix (column vector), 1-by-1 matrix (2-D scalar)
2-D	M-by-N matrix, 1-by-N matrix (row vector), M-by-1 matrix (column vector), 1-by-1 matrix (2-D scalar)	1-D vector, 1-D scalar
Scalar (1-D or 2-D)	1-D scalar, 1-by-1 matrix (2-D scalar)	1-D vector with length>1, M-by-N matrix with M>1 and/or N>1
Vector (1-D or 2-D)	1-D vector, 1-D scalar, 1-by-N matrix (row vector), M-by-1 matrix (column vector), 1-by-1 matrix (2-D scalar) Vector (1-D or 2-D) or scalar	M-by-N matrix with M>1 and N>1
Row Vector (2-D)	1-by-N matrix (row vector), 1-by-1 matrix (2-D scalar) Row vector (2-D) or scalar	1-D vector, 1-D scalar, M-by-N matrix with M>1
Column Vector (2-D)	M-by-1 matrix (column vector), 1-by-1 matrix (2-D scalar) Column vector (2-D) or scalar	1-D vector, 1-D scalar, M-by-N matrix with N>1

Dimensions (Continued)	Is...	Is not...
Full matrix	M-by-N matrix with M>1 and N>1	1-D vector, 1-D scalar, 1-by-N matrix (row vector), M-by-1 matrix (column vector), 1-by-1 matrix (2-D scalar)
Square matrix	M-by-N matrix with M=N, 1-D scalar, 1-by-1 matrix (2-D scalar)	M-by-N matrix with M≠N, 1-D vector, 1-by-N matrix (row vector), M-by-1 matrix (column vector)

Note that when **Signal dimensions** is selected from the model window **Format** menu, Simulink displays the size of a 1-D vector signal as an unbracketed integer, and displays the dimension of a 2-D signal as a pair of bracketed integers, `[MxN]`. Simulink *does not display* any size information for a 1-D or 2-D scalar signal. Dimension information for a signal can also be displayed in a model by attaching a Probe block with **Probe signal dimensions** selected.

- **Data type**

  Checks the signal data type for compliance (**Is...**) or noncompliance (**Is not...**) with the attributes in the subordinate **General data type** menu, which are shown in the table below. Any of the specific data types listed in the **Is...**

column below can be individually selected from the subordinate **Specific data type** menu.

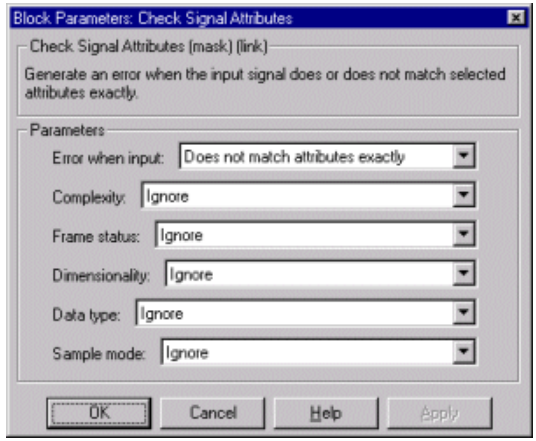
General data type	Is...	Is not...
Boolean	`boolean`	`single`, `double`, `uint8`, `int8`, `uint16`, `int16`, `uint32`, `int32`, fixed-point
Floating-point	`single`, `double`	`boolean`, `uint8`, `int8`, `uint16`, `int16`, `uint32`, `int32`, fixed-point
Fixed-point	fixed-point	`boolean`, `uint8`, `int8`, `uint16`, `int16`, `uint32`, `int32`, `single`, `double`
Integer	**Signed integer** `int8`, `int16`, `int32` **Unsigned integer** `uint8`, `uint16`, `uint32`	`boolean`, `single`, `double`

Note that data type information can also be displayed in a model by selecting **Port data types** from the model window's **Format** menu.

- **Sample mode**

  Checks whether the signal is discrete-time or continuous-time. (Note that when **Sample time colors** is selected from the **Format** menu, Simulink displays continuous-time signal lines in black or grey and discrete-time signal lines in colors corresponding to the relative rate. When a Probe block with **Probe sample time** enabled is attached to a continuous-time signal, the block icon displays the string `Ts:[0 x]`, where `x` is the sample time offset. When a Probe block is attached to a discrete-time signal, the block icon displays the string `Ts:[t 0]` for a sample-based signal or `Tf:[t 0]` for a frame-based signal, where `t` is the nonzero sample period or frame period, respectively. Frame-based signals are almost always discrete-time.)

Dialog Box



Error if input

Specifies whether the block generates an error when the input possesses *none* of the required attributes (**Does not match attributes exactly**), or when the input possesses *all* of the required attributes (**Matches attributes exactly**).

Complexity

The complexity for which the input should be checked, **Real** or **Complex**.

Frame status

The frame status for which the input should be checked, **Sample-based** or **Frame-based**.

Dimensionality

Specifies whether the input should be checked for compliance (**Is...**) or noncompliance (**Is not...**) with the attributes in the subordinate **Dimension** menu.

Dimensions

The dimensions for which the input should be checked. This parameter is available when **Is...** or **Is not...** is selected in the **Dimensionality** menu.

Data type

Specifies whether the input should be checked for compliance (**Is...**) or noncompliance (**Is not...**) with the attributes in the subordinate **General data type** menu.

General data type

The general data type for which the input should be checked. This parameter is available when **Is...** or **Is not...** is selected in the **Data type** menu, and enables the subordinate **Specific data type** parameter in most cases.

Specific data type

The specific data type for which the input should be checked. This parameter is available when **Floating-point**, **Fixed-point**, or **Integer** is selected in the **General data type** menu.

Sample mode

The sample mode for which the input should be checked, **Discrete** or **Continuous**.

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

Chirp

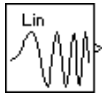
Purpose

Generate a swept-frequency cosine.

Library

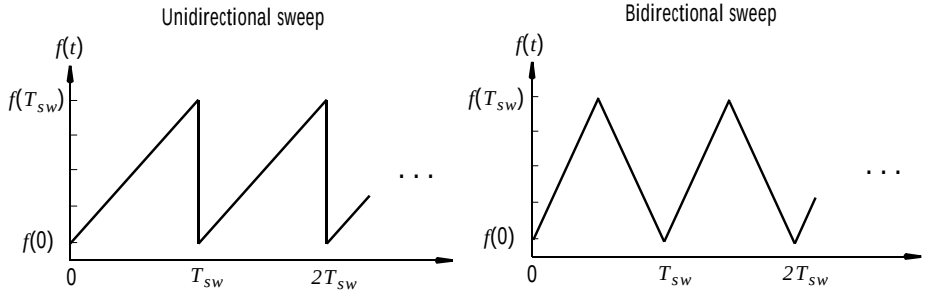
DSP Sources

Description



The Chirp block generates a unity-amplitude swept-frequency cosine (*chirp*) signal. The instantaneous output frequency is initialized to the **Initial frequency** parameter value, $f(0)$, and then varies continuously for the duration of the **Sweep period**. The curve of the transition between these two frequencies is specified by the **Frequency sweep** parameter, and can be **Swept cosine**, **Linear**, **Quadratic**, or **Logarithmic**.

When **Sweep mode** is set to **Unidirectional**, the cosine frequency is immediately reset to $f(0)$ after the **Sweep period** is traversed. Thus, the block repeats the unidirectional frequency sweep at the interval specified by the **Sweep period**, T_{sw}. When the **Sweep mode** is set to **Bidirectional**, the frequency sweep reverses direction half way through the period, and returns to $f(0)$ along a symmetrical trajectory. The block repeats the bidirectional frequency sweep at the interval specified by the **Sweep period** as well.



The method that the block uses to transition between the specified instantaneous frequencies is set by the **Frequency sweep** parameter, and can be **Swept cosine**, **Linear**, **Quadratic**, or **Logarithmic.** You must choose the **Target time**, t_g, and **Target frequency**, $f(t_g)$, to appropriately set the value for the constant term β in the formulas below. Note that the instantaneous frequency at the end of a frequency sweep is defined to be the value of the frequency sweep function at the sweep period, or $f(T_{sw})$.

- **Swept cosine** is similar to **Linear**, described below, but does not compensate for high frequencies introduced by short signal durations. As a result, the actual frequency content of the chirp signal may be far greater than the

frequency range specified by $f(t_0)$ and $f(T_{sw})$. The **Linear**, **Quadratic**, and **Logarithmic** frequency sweeps compensate for such signal transients; most of the frequencies generated by these methods lie within the range between $f(0)$ and $f(t_g)$.

- **Linear** uses an instantaneous frequency sweep $f(t)$ of

  $$f(t) = f(0) + \beta t$$

  where

  $$\beta = \frac{f(t_g) - f(0)}{t_g}$$

- **Quadratic** uses an instantaneous frequency sweep $f(t)$ of

  $$f(t) = f(0) + \beta t^2$$

  where β is the same as in the linear case.

- **Logarithmic** uses an instantaneous frequency sweep $f(t)$ of

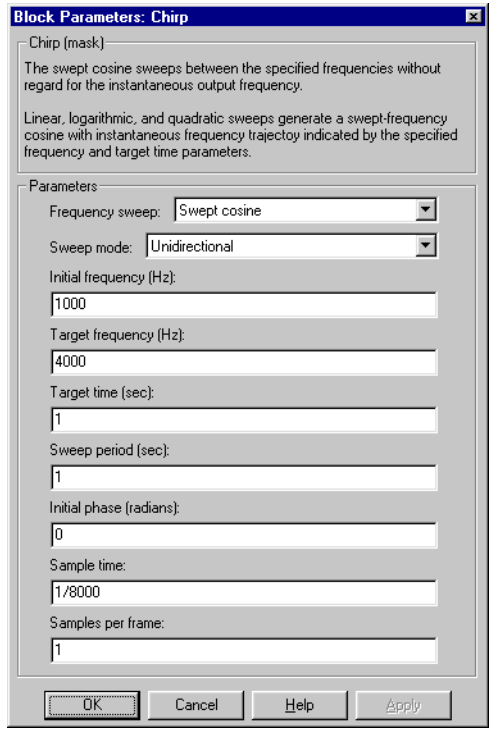
  $$f(t) = f(0) + 10^{\beta t}$$

  where

  $$\beta = \frac{\log[f(t_g) - f(0)]}{t_g}$$

  For the logarithmic sweep, the **Frequency at target time** parameter value must be greater than the **Initial frequency** parameter value. That is, $f(t_g) > f(0)$.

Dialog Box



Frequency sweep

The type of function that defines the instantaneous frequency trajectory. Tunable.

Sweep mode

The directionality of the chirp signal, **Unidirectional** or **Bidirectional**. Tunable.

Initial frequency (Hz)

The initial frequency, $f(0)$, of the output cosine. Tunable.

Target frequency (Hz)

The frequency, $f(t_g)$, that, along with the target time, t_g, defines the value of the constant β in the frequency sweep function. Tunable.

Target time (sec)

The target time t_g corresponding to the terminal frequency $f(t_g)$. Tunable.

Sweep period (sec)

The interval T_{sw} between successive repetitions of the frequency sweep. Tunable.

Initial phase (radians)

The phase of the cosine output at t=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.

See Also

Signal From Workspace	DSP Blockset
Signal Generator	Simulink
Sine Wave	DSP Blockset
`chirp`	Signal Processing Toolbox

See “Creating Signals Using Signal Generator Blocks” on page 3-36 for related information.

Cholesky Factorization

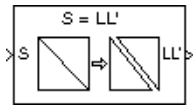
Purpose

Factor a square Hermitian positive definite matrix into triangular components.

Library

Math Functions / Matrices and Linear Algebra / Matrix Factorizations

Description

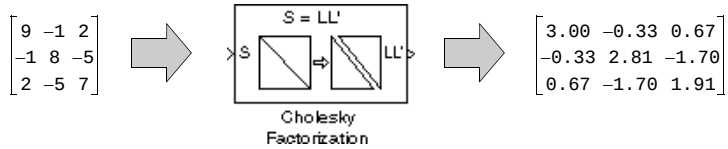


The Cholesky Factorization block uniquely factors the square Hermitian positive definite input matrix S as

$$S = LL^*$$

where L is a lower triangular square matrix with positive diagonal elements and L^* is the Hermitian (complex conjugate) transpose of L. Only the diagonal and upper 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 from L and upper triangle elements from L^*, and is always sample-based.



$$L = \begin{bmatrix} 3.00 & 0 & 0 \\ -0.33 & 2.81 & 0 \\ 0.67 & -1.70 & 1.91 \end{bmatrix}$$

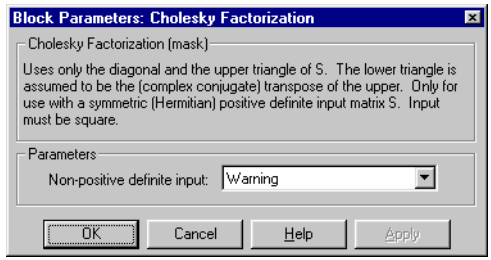
Note that L and L^* share the same diagonal in the output matrix. Cholesky factorization requires half the computation of Gaussian elimination (LU decomposition), and is always stable.

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.

Dialog Box



Non-positive definite input

Response to non-positive definite matrix inputs. Tunable.

References

Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

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 4-32 for related information.

Cholesky Inverse

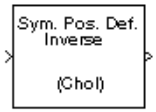
Purpose

Compute the inverse of a Hermitian positive definite matrix using Cholesky factorization.

Library

Math Functions / Matrices and Linear Algebra / Matrix Inverses

Description



The Cholesky Inverse block computes the inverse of the Hermitian positive definite input matrix S by performing Cholesky factorization.

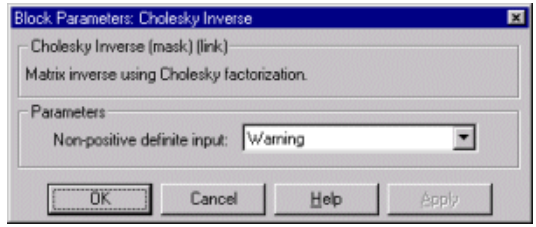
$$S^{-1} = (LL^*)^{-1}$$

L is a lower triangular square matrix with positive diagonal elements and L^* is the Hermitian (complex conjugate) transpose of L. Only the diagonal and upper triangle of the input matrix are used, and any imaginary component of the diagonal entries is disregarded. Cholesky factorization requires half the computation of Gaussian elimination (LU decomposition), and is always stable. The output is always sample-based.

The algorithm requires that the input be Hermitian positive definite. When the input is not positive definite, the block reacts with the behavior specified by the **Non-positive definite input** parameter. The following options are available:

- **Ignore** – Proceed with the computation and *do not* issue an alert. The output is *not* a valid inverse.
- **Warning** – Display a warning message in the MATLAB command window, and continue the simulation. The output is *not* a valid inverse.
- **Error** – Display an error dialog box and terminate the simulation.

Dialog Box



Non-positive definite input

Response to non-positive definite matrix inputs. Tunable.

References

Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

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 4-34 for related information.

Cholesky Solver

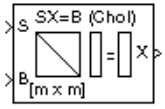
Purpose

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-hand side M-by-N matrix, B. The output is the unique solution of the equations, M-by-N matrix *X*, and is always sample-based.

When the input is not positive definite, the block reacts with the behavior specified by the **Non-positive definite input** parameter. The following options are available:

- **Ignore** – Proceed with the computation and *do not* issue an alert. The output is *not* a valid solution.
- **Warning** – Proceed with the computation and display a warning message in the MATLAB command window. The output is *not* a valid solution.
- **Error** – Display an error dialog box and terminate the simulation.

A length-M vector input for right-hand side B is treated as an M-by-1 matrix.

Algorithm

Cholesky factorization uniquely factors the Hermitian positive definite input matrix S as

$$S = LL^*$$

where L is a lower triangular square matrix with positive diagonal elements.

The equation S*X*=B then becomes

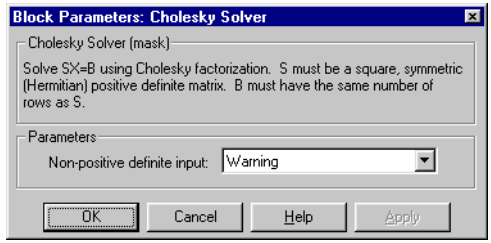
$$LL^*X = B$$

which is solved for *X* by making the substitution $Y = L^*X$, and solving the following two triangular systems by forward and backward substitution, respectively.

$$LY = B$$

$$L^{*}X = Y$$

Dialog Box



Non-positive definite input

Response to non-positive definite matrix inputs. Tunable.

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 4-31 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 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 input is altered by the application of a linear phase term so that there is no phase discontinuity at $\pm\pi$ radians. That is, each input channel is independently zero padded and circularly shifted to have zero phase at π radians.

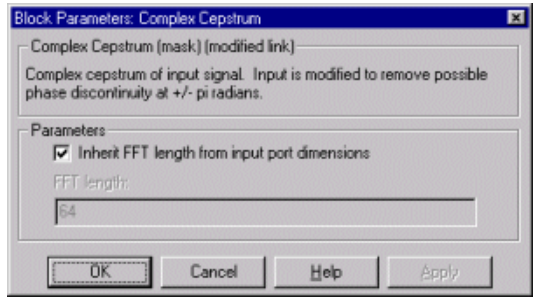
The output is a real M_o-by-N matrix, where M_o is specified by the **FFT length** parameter. Each output column contains the length-M_o complex cepstrum of the corresponding input column.

```
y = cceps(u,Mo)    % Equivalent MATLAB code
```

When the **Inherit FFT length from input port dimensions** check box is selected, the output frame size matches the input frame size (M_o=M). In this case, a *sample-based* length-M row vector input is processed as a single channel (i.e., as an M-by-1 column vector), and the output is a length-M row vector. A 1-D vector input is *always* processed as a single channel, and the output is a 1-D vector.

The output is always sample-based, and the output port rate is the same as the input port rate.

Dialog Box



Inherit FFT length from input port dimensions

When selected, matches the output frame size to the input frame size.

FFT length

The number of frequency points at which to compute the FFT, which is also the output frame size, M_o. This parameter is available when **Inherit FFT length from input port dimensions** is not selected.

See Also

DCT	DSP Blockset
FFT	DSP Blockset
Real Cepstrum	DSP Blockset
`cceps`	Signal Processing Toolbox

Complex Exponential

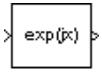
Purpose

Compute the complex exponential function.

Library

Math Functions / Math Operations

Description

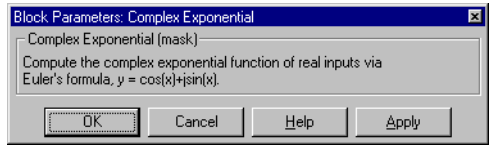


The Complex Exponential block computes the complex exponential function for each element of the real input, u.

$$y = e^{ju} = \cos u + j \sin u$$

where $j = \sqrt{-1}$. The output is complex, with the same size and frame status as the input.

Dialog Box



See Also

Math Function
Sine Wave
exp
Simulink
DSP Blockset
MATLAB

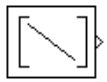
Purpose

Generate a square, diagonal matrix.

Library

DSP Sources,
Math Functions / Matrices and Linear Algebra / Matrix Operations

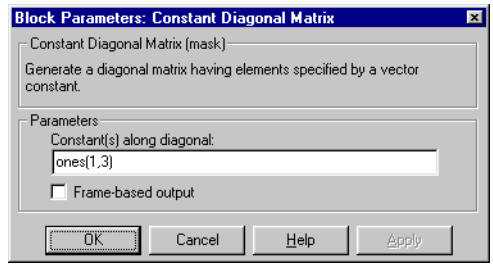
Description



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



Constant(s) along diagonal

The values of the elements along the diagonal, as a scalar or vector. Tunable.

Frame-based output

Specifies frame-based output when selected.

See Also

Create Diagonal Matrix	DSP Blockset
DSP Constant	DSP Blockset
Identity Matrix	DSP Blockset
`diag`	MATLAB

See “Creating Signals Using Constant Blocks” on page 3-33 for related information.

Constant Ramp

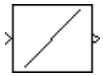
Purpose

Generate a ramp signal with length based on input dimensions.

Library

DSP Sources

Description



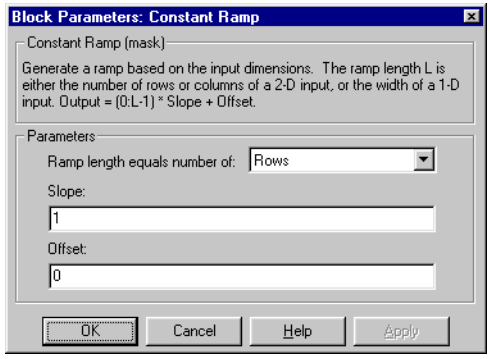
The Constant Ramp block generates the constant ramp signal

```
y = (0:L-1)*m + b
```

where `m` is the slope specified by the scalar **Slope** parameter, `b` is the *y*-intercept specified by the scalar **Offset** parameter.

For a matrix input, the length `L` of the output ramp is equal to either the number of rows or the number of columns in the input, as determined by the **Ramp length equals number of** parameter. For a 1-D vector input, `L` is equal to the length of the input vector. The output, `y`, is always a 1-D vector.

Dialog Box



Ramp length equals number of

The dimension of the input matrix that determines the length of the output ramp, **Rows** or **Columns**.

Slope

The slope of the ramp, a scalar.

Offset

The *y*-intercept of the ramp, a scalar.

See Also

Create Diagonal Matrix	DSP Blockset
Identity Matrix	DSP Blockset
DSP Constant	DSP Blockset

See “Creating Signals Using Constant Blocks” on page 3-33 for related information.

Contiguous Copy

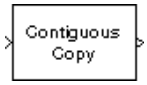
Purpose

Create a discontiguous input in a contiguous block of memory (for RTW code generation from blocks linked to versions of the DSP Blockset prior to 4.0).

Library

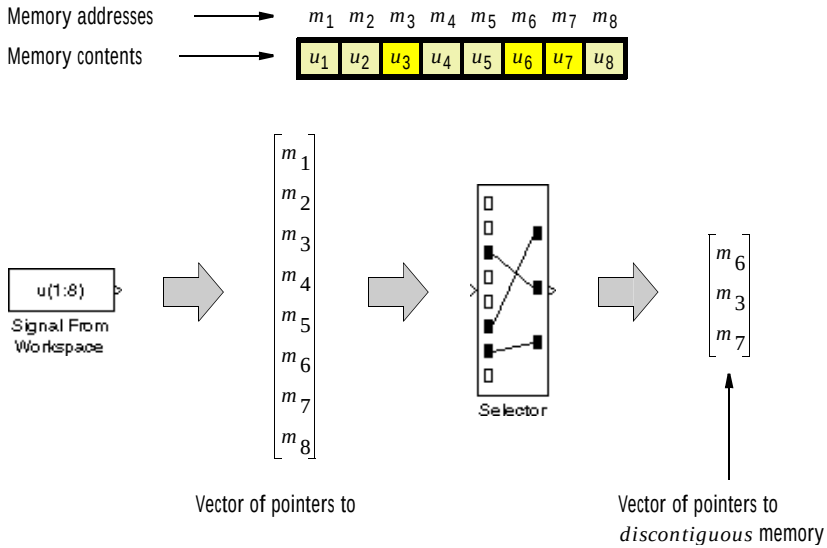
Signal Management / Signal Attributes

Description

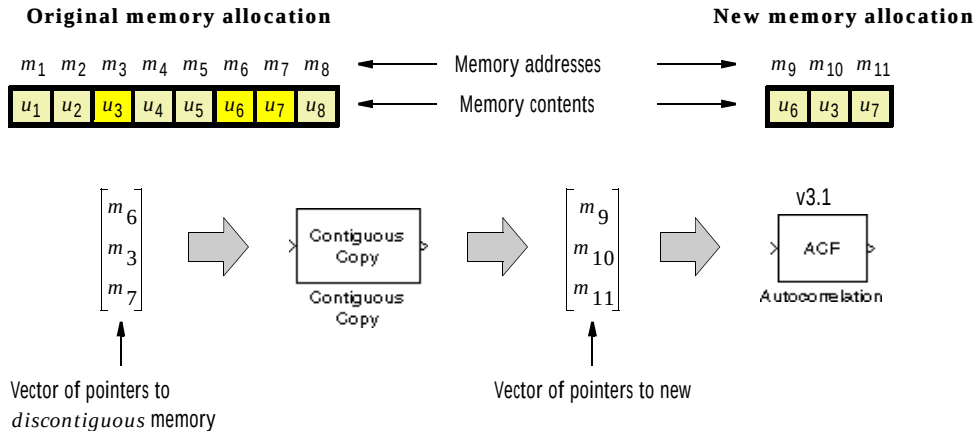


The Contiguous Copy block copies the input to a contiguous block of memory, and passes this new copy to the output. The output is identical to the input, but is guaranteed to reside in a contiguous section of memory.

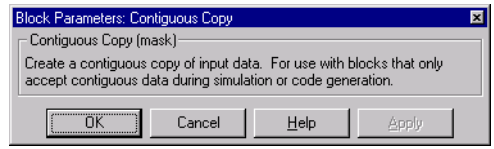
Because Simulink employs an efficient copy-by-reference method for propagating data in a model, some operations produce outputs with discontiguous memory locations. An example of such an operation is shown below with the Simulink Selector block.



Although this does not present a problem during simulation, blocks linked to versions of the DSP Blockset prior to 4.0 may require contiguous inputs for code-generation with the Real-Time Workshop (RTW). When such blocks are used in a model intended for code generation, they should be preceded by the Contiguous Copy block to ensure that their inputs are contiguous. The DSP Blockset version 3.1 Autocorrelation block shown below is an example of one that requires contiguous inputs for code generation.


Original memory allocation
New memory allocation
m1 m2 m3 m4 m5 m6 m7 m8
u1 u2 u3 u4 u5 u6 u7 u8
Memory addresses
Memory contents
m9 m10 m11
u6 u3 u7
m6 m3 m7
Contiguous Copy
Contiguous Copy
m9 m10 m11
v3.1
ACF
Autocorrelation
Vector of pointers to discontiguous memory
Vector of pointers to new

Dialog Box


Block Parameters: Contiguous Copy
Contiguous Copy (mask)
Create a contiguous copy of input data. For use with blocks that only accept contiguous data during simulation or code generation.
OK
Cancel
Help
Apply

Convert 1-D to 2-D

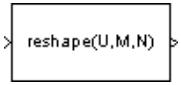
Purpose

Reshape a 1-D or 2-D input to a 2-D matrix with the specified dimensions.

Library

Signal Management / Signal Attributes

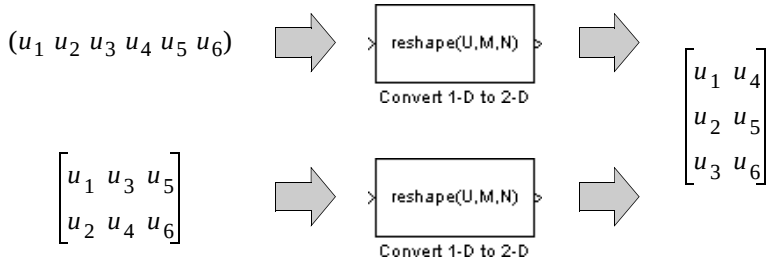
Description



The Convert 1-D to 2-D block reshapes a length-M_i 1-D vector or an M_i-by-N_i matrix to an M_o-by-N_o matrix, where M_o is specified by the **Number of output rows** parameter, and N_o is specified by the **Number of output columns** parameter.

```
y = reshape(u,Mo,No)        % Equivalent MATLAB code
```

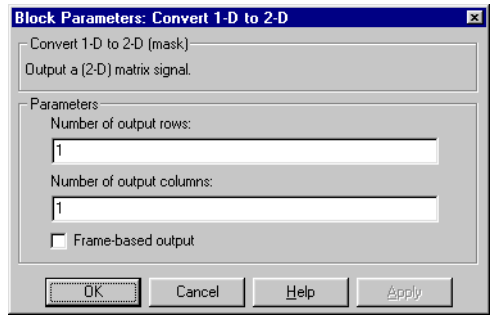
The input is reshaped *columnwise*, as shown in the two cases below. The length-6 vector and the 2-by-3 matrix are both reshaped to the same 3-by-2 output matrix.



An error is generated if $(M_o*N_o) \neq (M_i*N_i)$. That is, the total number of input elements must be conserved in the output.

The output is frame-based if the **Frame-based output** check box is selected; otherwise, the output is sample-based.

Dialog Box



Number of output rows

The number of rows, M_o, in the output matrix.

Number of output columns

The number of rows, N_o, in the output matrix.

Frame-based output

Creates a frame-based output when selected.

See Also

Buffer	DSP Blockset
Convert 2-D to 1-D	DSP Blockset
Frame Status Conversion	DSP Blockset
Reshape	Simulink
Submatrix	DSP Blockset

Convert 2-D to 1-D

Purpose

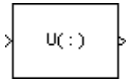
Convert a 2-D matrix input to a 1-D vector.

Library

Signal Management / Signal Attributes

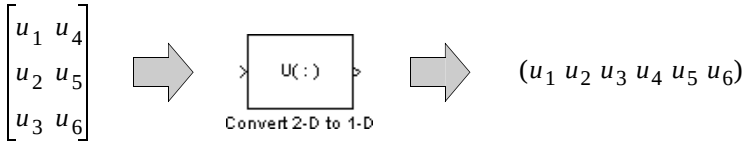
Description

The Convert 2-D to 1-D block reshapes an M-by-N matrix input to a 1-D vector with length M*N.



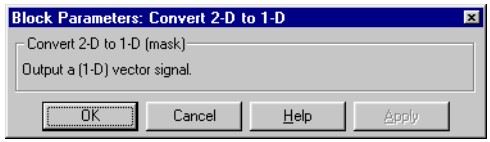

```
y = u(:)          % Equivalent MATLAB code
```

The input is reshaped *columnwise*, as shown below for a 3-by-2 matrix.



The output is always sample-based.

Dialog Box



See Also

Buffer	DSP Blockset
Convert 1-D to 2-D	DSP Blockset
Frame Status Conversion	DSP Blockset
Reshape	Simulink
Submatrix	DSP Blockset

Purpose

Convert complex data from the DSP Blockset Version 2.2 format to the Simulink Version 3 format.

Library

Elementary Functions, in Math Functions

Description

The Convert Complex DSP To Simulink block accepts complex data (scalar, vector, matrix) in the DSP Blockset Version 2.2 format, and outputs the same data in the Simulink Version 3 complex format. Only complex data should be supplied to this block.

Blocks provided in Release 11 and later blocksets (e.g., Simulink Version 3.0, DSP Blockset Version 3.0, Fixed Point Blockset Version 2.0) use the Simulink Version 3 complex format, which is not compatible with the DSP Blockset Version 2.2 complex format. To add a new block or subsystem (Release 11 and later) to an existing model that uses the DSP Blockset Version 2.2 complex data format, precede it with the Convert Complex DSP To Simulink block. If the new block or subsystem's output is complex, you should follow it with the complementary Convert Complex Simulink To DSP block (unless the downstream blocks have already been updated to their Release 11 counterparts).

These convertor blocks are only needed for interfacing Version 3.0 blocks to the *complex-data* section of a Version 2.2 or earlier model. Version 3.0 blocks can be added to *real-data* sections of older models without any data format conversion.

Note Within a section of model that uses the Version 2.2 complex format, you should continue to use the complex port identifier (∗) as a guide to wiring blocks. Outputs ports labeled with the ∗ symbol should only be connected to input ports labeled with the ∗ symbol.

The following figure shows how you can use these two convertor blocks to migrate part of a complex-data model to the Version 3.0 complex format while letting other components continue to use the Version 2.2 complex-data format.

Existing (Version 2.2) complex-data

Subsystem B upgraded to Version 3.0 complex-data

(Subsystem A remains a Version 2.2 implementation)

Subsystem A upgraded to Version 3.0 complex-data

(Subsystem B remains a Version 2.2 implementation)

Dialog Box

See Also

Convert Complex Simulink To DSP DSP Blockset

Purpose

Convert complex data from the Simulink Version 3 format to the DSP Blockset Version 2.2 format.

Library

Elementary Functions, in Math Functions

Description

The Convert Complex Simulink To DSP block accepts complex data (scalar, vector, matrix) in the Simulink Version 3 format, and outputs the same data in the DSP Blockset Version 2.2 complex format. Only complex data should be supplied to this block.

Blocks provided in Release 11 and later blocksets (e.g., Simulink Version 3.0, DSP Blockset Version 3.0, Fixed Point Blockset Version 2.0) use the Simulink Version 3 complex format, which is not compatible with the DSP Blockset Version 2.2 complex format. To add a new block or subsystem (Release 11 and later) to an existing model that uses the DSP Blockset Version 2.2 complex data format, precede it with the Convert Complex DSP To Simulink block. If the new block's output is complex, you should then follow it with the Convert Complex Simulink To DSP block (unless the downstream blocks have already been updated to their Release 11 counterparts).

These convertor blocks are only needed for interfacing Version 3.0 blocks to the *complex-data* section of a Version 2.2 or earlier model. Version Version 3.0 blocks can be added to *real-data* sections of older models without any data format conversion.

Note Within a section of model that uses the Version 2.2 complex format, you should continue to use the complex port identifier (∗) as a guide to wiring blocks. Outputs ports labeled with the ∗ symbol should only be connected to input ports labeled with the ∗ symbol.

The following figure shows how you can use these two convertor blocks to migrate part of a complex-data model to the Version 3.0 complex format while letting other components continue to use the Version 2.2 complex-data format.

Convert Complex Simulink To DSP

Existing (Version 2.2) complex-data

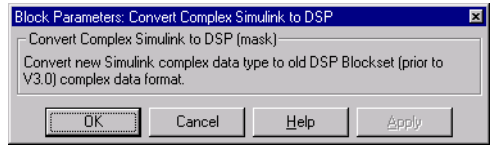
Subsystem B upgraded to Version 3.0 complex-data

(Subsystem A remains a Version 2.2 implementation)

Subsystem A upgraded to Version 3.0 complex-data

(Subsystem B remains a Version 2.2 implementation)

Dialog Box



See Also

Convert Complex DSP To Simulink DSP Blockset

Purpose

Compute the convolution of two inputs.

Library

Signal Operations

Description

The Convolution block convolves corresponding columns (channels) of M_u-by-N input matrix u and M_v-by-N input matrix v.

Frame-Based Inputs

Matrix inputs must be frame-based. The output, y, is a frame-based (M_u+M_v-1)-by-N matrix whose jth column has elements

$$y_{i,j} = \sum_{k=1}^{\max(M_u, M_v)} u_{k,j} v^*_{(i-k+1),j} \qquad 1 \le i \le (M_u + M_v - 1)$$

where $*$ denotes the complex conjugate. Inputs u and v are zero when indexed outside of their valid ranges. When both inputs are real, the output is real; when one or both inputs are complex, the output is complex.

When one input is a column vector (single channel) and the other is a matrix (multiple channels), the single-channel input is independently convolved with each channel of the multichannel input. For example, if u is a M_u-by-1 column vector and v is an M_v-by-N matrix, the output is an (M_u+M_v-1)-by-N matrix whose jth column has elements

$$y_{i,j} = \sum_{k=1}^{\max(M_u, M_v)} u_k v^*_{(i-k+1),j} \qquad 1 \le i \le (M_u + M_v - 1)$$

Sample-Based Inputs

If u and v are sample-based vectors with lengths M_u and M_v, the Convolution block performs the vector convolution

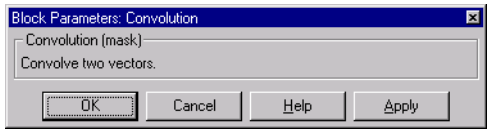
$$y_i = \sum_{k=1}^{\max(M_u, M_v)} u_k v^*_{(i-k+1)} \qquad 1 \le i \le (M_u + M_v - 1)$$

The dimensions of the sample-based output vector are determined by the dimensions of the input vectors:

- When both inputs are row vectors, or when one input is a row vector and the other is a 1-D vector, the output is a 1-by-(M_u+M_v-1) row vector.
- When both inputs are column vectors, or when one input is a column vector and the other is a 1-D vector, the output is a (M_u+M_v-1)-by-1 column vector.
- When both inputs are 1-D vectors, the output is a 1-D vector of length M_u+M_v-1.

The Convolution block does not accept sample-based full-dimension matrix inputs, or mixed sample-based row vector and column vector inputs.

Dialog Box



See Also

Correlation — DSP Blockset

conv — MATLAB

Purpose

Compute the correlation along the columns of two inputs.

Library

Statistics

Description

The Correlation block computes the cross-correlation of corresponding columns (channels) of the M_u-by-N input matrix u and M_v-by-N input matrix v. The frame status of both inputs must be the same. The block does not accept sample-based full-dimension matrix inputs or 2-D row vector inputs.

Frame-Based Inputs

Matrix inputs must be frame-based. The output, y, is a frame-based (M_u+M_v-1)-by-N matrix whose jth column has elements

$$y_{i,j} = \sum_{k=1}^{\max(M_u, M_v)} u_{k,j} v^*_{(k+i-M_v),j} \qquad 1 \le i \le (M_u + M_v - 1)$$

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 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

$$y_{i,j} = \sum_{k=1}^{\max(M_u, M_v)} u_k v^*_{(k-i+M_v),j} \qquad 1 \le i \le (M_u + M_v - 1)$$

Sample-Based Inputs

Matrix inputs cannot be sample based, so all sample-based inputs are column vectors or 1-D vectors. (the block does not support 2-D row vector inputs.) If u and v are sample-based vectors with lengths M_u and M_v, the Correlation block performs the vector cross-correlation

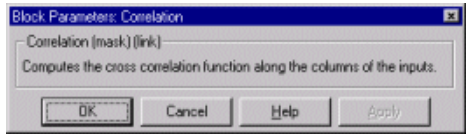
$$y_i = \sum_{k=1}^{\max(M_u, M_v)} u_k v^*_{(k-i+M_v)} \qquad 1 \le i \le (M_u + M_v - 1)$$

The dimensions of the sample-based output vector are determined by the dimensions of the input vectors:

- When both inputs are column vectors, or when one input is a column vector and the other is a 1-D vector, the output is a (M_u+M_v-1)-by-1 column vector.
- When both inputs are 1-D vectors, the output is a 1-D vector of length M_u+M_v-1.

The Correlation block does not accept sample-based full-dimension matrix inputs or 2-D row vector inputs.

Dialog Box



See Also

Autocorrelation	DSP Blockset
Convolution	DSP Blockset
`xcorr`	Signal Processing Toolbox

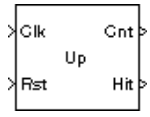
Purpose

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 (i.e., single channel). If both inputs are sample-based, they must have the same sample period. If the `Clk` input is frame-based, the frame period must equal the sample period of the `Rst` input.

The trigger event for both inputs is specified by the **Count event** pop-up menu, and can be one of the following:

- **Rising edge** triggers a count or reset operation when the `Clk` or `Rst` input rises from a negative value to zero or a positive value, or from zero to a positive value.
- **Falling edge** triggers a count or reset operation when the `Clk` or `Rst` input falls from a positive value to zero or a negative value, or from zero to a negative value.
- **Either edge** triggers a count or reset operation when either a rising or falling edge (as described above) occurs.
- **Nonzero sample** triggers a count or reset operation at each sample time that 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. See “Free-Running Operation” below.

At the start of the simulation, the block sets the counter to the value specified by the **Initial count** parameter, which can be any integer in the range defined by the **Counter size** parameter. The **Counter size** parameter allows you to choose from three standard counter ranges, or to specify an arbitrary counter limit:

- **8 bits** specifies a counter with a range of 0 to 255.

- **16 bits** specifies a counter with a range of 0 to 65535.
- **32 bits** specifies a counter with a range of 0 to 2^{32}-1.
- **User defined** enables the supplementary **Maximum count** parameter, which allows you to specify an arbitrary integer as the upper count limit. The range of the counter is then 0 to the **Maximum count** value.

Sample-Based Operation

The block operates in sample-based mode when the `Clk` input is a sample-based scalar. Sample-based vectors and matrices are not accepted.

When the **Count direction** parameter is set to **Up**, a sample-based trigger event at the `Clk` input causes the block to increment the counter by one. The block continues incrementing the counter when triggered until the counter value reaches the upper count limit (e.g., 255 for an 8-bit counter). At the next `Clk` trigger event, the block resets the counter to 0, and resumes incrementing the counter with the subsequent `Clk` trigger event.

When the **Count direction** parameter is set to **Down**, a sample-based trigger event at the `Clk` input causes the block to decrement the counter by one. The block continues decrementing the counter when triggered until the counter value reaches 0. At the next `Clk` trigger event, the block resets the counter to the upper count limit (e.g., 255 for an 8-bit counter), and resumes decrementing the counter with the subsequent `Clk` trigger event.

Between triggering events the block holds the output at its most recent value. The block resets the counter to its initial state when the trigger event specified in the **Count event** menu is received at the optional `Rst` input. When trigger events are received simultaneously at the `Clk` and `Rst` ports, the block first resets the counter, and then increments or decrements appropriately. (If you do not need to reset the counter during the simulation, you can disable the `Rst` port by deselecting the **Reset input** check box.)

The **Output** pop-up menu provides three options for the output port configuration of the block icon:

- **Count** configures the block icon to show a `Cnt` port, which produces the current value of the counter as a sample-based scalar with the same sample period as the inputs.
- **Hit** configures the block icon to show a `Hit` port. The `Hit` port produces zeros while the value of the counter does not equal the integer **Hit value**

parameter setting. When the counter value *does* equal the **Hit value** setting, the block generates a value of `1` at the `Hit` port. The output is sample-based with the same sample period as the inputs.

- **Count and Hit** configures the block icon with both ports.

Frame-Based Operation

The block operates in frame-based mode when the `Clk` input is a frame-based vector (i.e., single channel). Multichannel frame-based inputs are not accepted.

Frame-based operation is the same as sample-based operation, except that the block increments or decrements the counter by the total number of trigger events contained in the `Clk` input frame. A trigger event that is split across two consecutive frames is counted in the frame that contains the conclusion of the event. When a trigger event is received at the `Rst` port, the block first resets the counter, and then increments or decrements the counter by the number of trigger events contained in the `Clk` frame.

The `Cnt` and `Hit` outputs are sample-based scalars with sample period equal to the `Clk` input frame period.

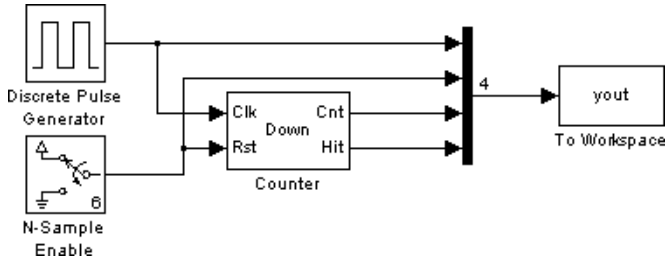
Free-Running Operation

The block operates in free-running mode when **Free running** is selected from the **Count event** menu.

The `Clk` input port is disabled in this mode, and the block simply increments or decrements the counter using the constant sample period specified by the **Sample time** parameter, T_s. The `Cnt` output is a frame-based M-by-1 matrix containing the count value at each of M consecutive sample times, where M is specified by the **Samples per output frame** parameter. The `Hit` output is a frame-based M-by-1 matrix containing the hit status (`0` or `1`) at each of those M consecutive sample times. Both outputs have a frame period of $M*T_s$.

Example

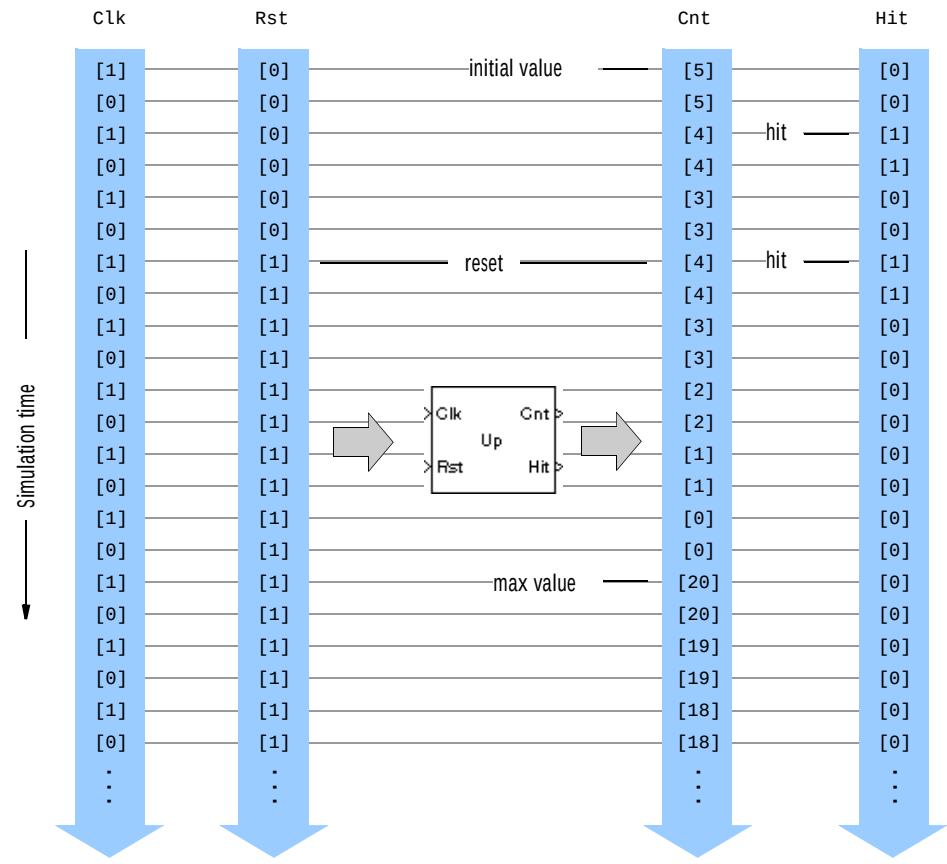
In the model below, the `Clk` port of the Counter block is driven by Simulink's Discrete Pulse Generator block, and the `Rst` port is triggered by an N-Sample Enable block. All of the Counter block's inputs and outputs are multiplexed into a single To Workspace block using a 4-port Mux block.



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 Discrete Pulse Generator and To Workspace blocks.)

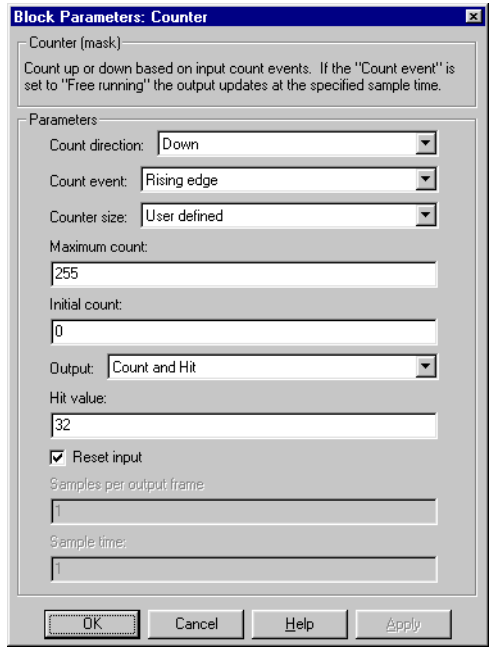
- Set the N-Sample Enable block parameters as follows:
  - **Trigger count** = `6`
  - **Active level** = **High (1)**
- Set the Counter block parameters as follows:
  - **Count direction** = **Down**
  - **Count event** = **Rising edge**
  - **Counter size** = **User defined**
  - **Maximum count** = `20`
  - **Initial count** = `5`
  - **Output** = **Count and Hit**
  - **Hit value** = `4`
  - **Reset input** ☑
- Set the **Number of inputs** parameter of the Mux block to `4`.

The figure below shows the first 22 samples of the model's four-column output, `yout`. The first column is the Counter block's `Clk` input, the second column is the block's `Rst` input, the third column is the block's `Cnt` output, and the fourth column is the block's `Hit` output.



You can see that the seventh input samples to both the `Clk` and `Rst` ports of the Counter block represent trigger events (rising edges), so at this time step the block first resets the counter to its initial value of 5, and then immediately decrements the count to 4. When the counter reaches its minimum value of 0, it rolls over to its maximum value of 20 with the following trigger event at the `Cnt` port.

Dialog Box



Count direction

The counter direction, **Up** or **Down**. Tunable, except in Simulink's 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.

Counter size

The range of integer values the block should count through before recycling to zero.

Maximum count

The counter's maximum value when **Counter size** is set to **User defined**.

Initial count

The counter's initial value at the start of the simulation and after reset. Tunable, except in Simulink's 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 Simulink's 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.

See Also

Edge Detector	DSP Blockset
N-Sample Enable	DSP Blockset
N-Sample Switch	DSP Blockset

Covariance AR Estimator

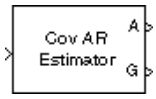
Purpose

Compute an estimate of AR model parameters using the covariance method.

Library

Estimation / Parametric Estimation

Description



The Covariance AR Estimator block uses the covariance method to fit an autoregressive (AR) model to the input data. This method minimizes the forward prediction error in the least-squares sense.

The input is a sample-based vector (row, column, or 1-D) or frame-based vector (column only) representing a frame of consecutive time samples from a single-channel signal, which is assumed to be the output of an AR system driven by white noise. The block computes the normalized estimate of the AR system parameters, $A(z)$, independently for each successive input frame.

$$H(z) = \frac{G}{A(z)} = \frac{G}{1 + a(2)z^{-1} + \ldots + a(p+1)z^{-p}}$$

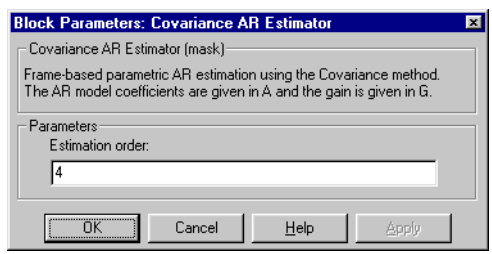
The order, p, of the all-pole model is specified by the **Estimation order** parameter.

The top output, `A`, is a column vector of length p+1 with the same frame status as the input, and contains the normalized estimate of the AR model coefficients in descending powers of z,

```
[1 a(2) ... a(p+1)]
```

The scalar gain, G, is provided at the bottom output (`G`).

Dialog Box



Estimation order

The order of the AR model, p.

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.

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

Covariance Method

Purpose

Compute a parametric spectral estimate using the covariance method.

Library

Estimation / Power Spectrum Estimation

Description

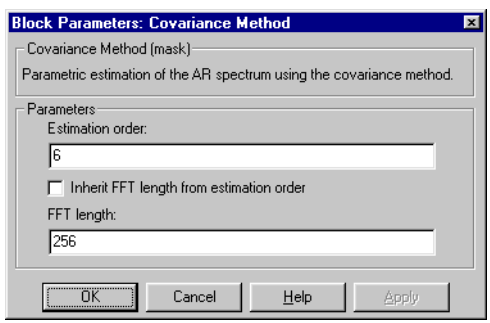
The Covariance Method block estimates the power spectral density (PSD) of the input using the covariance method. This method fits an autoregressive (AR) model to the signal by minimizing the forward prediction error in the least-squares sense. The order of the all-pole model is the value specified by the **Estimation order** parameter, and the spectrum is computed from the FFT of the estimated AR model parameters.

The input is a sample-based vector (row, column, or 1-D) or frame-based vector (column only) representing a frame of consecutive time samples from a single-channel signal. The block's output (a column vector) is the estimate of the signal's power spectral density at N_{fft} equally spaced frequency points in the range $[0,F_s)$, where F_s is the signal's sample frequency.

When **Inherit FFT length from input dimensions** is selected, N_{fft} is specified by the frame size of the input, which must be a power of 2. When **Inherit FFT length from input dimensions** is *not* selected, N_{fft} is specified as a power of 2 by the **FFT length** parameter, and the block zero pads or truncates the input to N_{fft} before computing the FFT. The output is always sample-based.

See the Burg Method block reference for a comparison of the Burg Method, Covariance Method, Modified Covariance Method, and Yule-Walker Method blocks.

Dialog Box



Estimation order

The order of the AR model.

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.

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.

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 4-30 for related information.

Create Diagonal Matrix

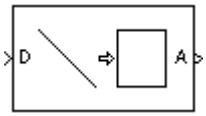
Purpose

Create a square diagonal matrix from diagonal elements.

Library

Math Functions / Matrices and Linear Algebra / Matrix Operations

Description

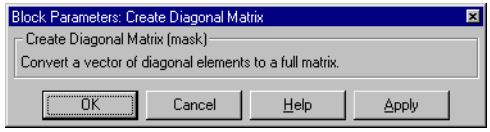


The Create Diagonal Matrix block populates the diagonal of the M-by-M matrix output with the elements contained in the length-M vector input, `D`. The elements off the diagonal are zero.

```
A = diag(D)        Equivalent MATLAB code
```

The output is always sample-based.

Dialog Box



See Also

Constant Diagonal Matrix	DSP Blockset
Extract Diagonal	DSP Blockset
`diag`	MATLAB

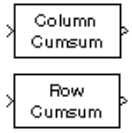
Purpose

Compute the cumulative sum of row or column elements.

Library

Math Functions / Math Operations

Description



The Cumulative Sum block computes the cumulative sum of the row or column elements in the M-by-N input matrix u.

Columnwise Summing

When the **Cumulative Sum along** parameter is set to **Columns**, the block computes the cumulative sum of the column elements.

```
y = cumsum(u)          % Equivalent MATLAB code
```

The output is an M-by-N matrix whose *j*th column has elements

$$y_{i,j} = \sum_{k=1}^{i} u_{k,j} \qquad 1 \le i \le M$$

The frame status of the output is the same as the input. For sample-based inputs, the first row of each successive output is the same as that of the corresponding input. For convenience, length-M 1-D vector inputs are treated as M-by-1 column vectors for column-wise summation, and the output is a length-M 1-D vector.

For frame-based inputs, the first row of the first output is the same as that of the first input, and the first row of each subsequent output contains the sum of the first row of the current input (time *t*) and the last row of the previous output (time t-T_f).

$$y_{1,j}(t) = u_{1,j}(t) + y_{M,j}(t - T_f)$$

Rowwise Summing

When the **Cumulative Sum along** parameter is set to **Rows**, the block computes the cumulative sum of the row elements.

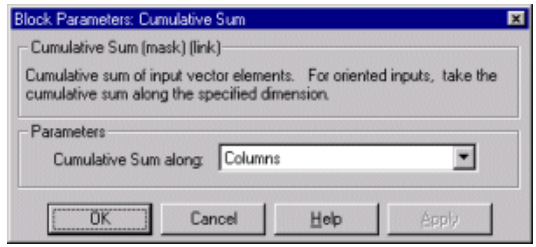
```
y = cumsum(u,2)        % Equivalent MATLAB code
```

The output 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 frame status of the output is the same as the input. For both sample-based and frame-based inputs, the first column of each successive output is the same as that of the corresponding input. For convenience, length-N 1-D vector inputs are treated as 1-by-N row vectors for row-wise summation, and the output is a length-N 1-D vector.

Dialog Box



Cumulative Sum along

The dimension along which to compute the cumulative summations. **Columns** specifies columnwise summation, while **Rows** specifies rowwise summation.

See Also

Difference	DSP Blockset
Matrix Sum	DSP Blockset
`cumsum`	MATLAB

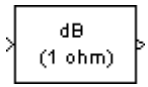
Purpose

Convert magnitude data to decibels (dB or dBm).

Library

Math Functions / Math Operations

Description



The dB Conversion block converts a linearly scaled power or amplitude input to dB or dBm. The **Input signal** parameter specifies whether the input is a power signal or a voltage signal, and the **Convert to** parameter controls the scaling of the output. When selected, the **Add eps to input to protect against "log(0) = -inf"** parameter adds a value of `eps` to all power and voltage inputs. When this option is not enabled, zero-valued inputs produce `-inf` at the output. The size and frame status of the output are the same as the input.

Power Inputs

Select **Power** as the **Input signal** parameter when the input, `u`, is a real, nonnegative, power signal (units of watts). When the **Convert to** parameter is set to **dB**, the block performs the dB conversion

```
y = 10*log10(u)            % Equivalent MATLAB code
```

When the **Convert to** parameter is set to **dBm**, the block performs the dBm conversion

```
y = 10*log10(u) + 30
```

The dBm conversion is equivalent to performing the dB operation *after* converting the input to milliwatts.

Voltage Inputs

Select **Amplitude** as the **Input signal** parameter when the input, `u`, is a real voltage signal (units of volts). The block uses the scale factor specified in ohms by the **Load resistance** parameter, `R`, to convert the voltage input to units of power (watts) before converting to dB or dBm. When the **Convert to** parameter is set to **dB**, the block performs the dB conversion

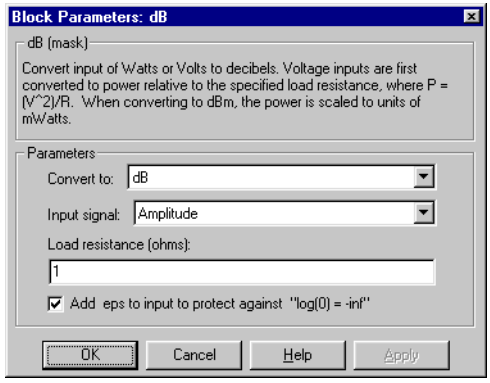
```
y = 10*log10(abs(u)^2/R)
```

When the **Convert to** parameter is set to **dBm**, the block performs the dBm conversion

```
y = 10*log10(abs(u)^2/R) + 30
```

The dBm conversion is equivalent to performing the dB operation *after* converting the `(abs(u)^2/R)` result to milliwatts.

Dialog Box



Convert to

The logarithmic scaling to which the input is converted, **dB** or **dBm**.

Input signal

The type of input signal, **Power** or **Amplitude**.

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).

See Also

dB Gain
Math Function
`log10`
DSP Blockset
Simulink
MATLAB

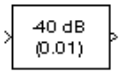
Purpose

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.

$$y_{ij} = 10 u_{ij}^{(g_{ij}/k)}$$

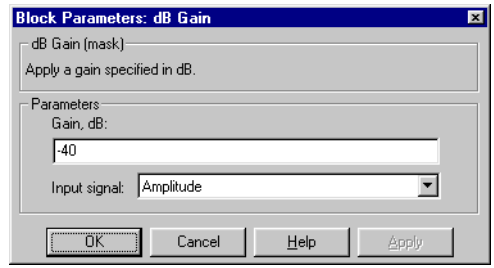
The value of k is 10 for power signals (select **Power** as the **Input signal** parameter) and 20 for voltage signals (select **Amplitude** as the **Input signal** parameter).

The value of the equivalent linear gain

$$g_{ij}^{lin} = 10^{(g_{ij}/k)}$$

is displayed in the block icon below the dB gain value. The size and frame status of the output are the same as the input.

Dialog Box



Gain

The dB gain to apply to the input, a scalar or a real M-by-N matrix. Tunable.

Input signal

The type of input signal: **Power** or **Amplitude**. Tunable.

See Also

dB Conversion
Math Function
`log10`
DSP Blockset
Simulink
MATLAB

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.

$$y(k,l) = w(k)\sum_{m=1}^{M} u(m,l)\cos\frac{\pi(2m-1)(k-1)}{2M}, \qquad k = 1, \ldots, M$$

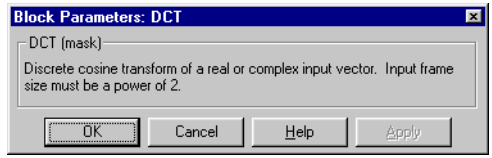
where

$$w(k) = \begin{cases} \frac{1}{\sqrt{M}}, & k = 1 \\ \sqrt{\frac{2}{M}}, & 2 \le k \le M \end{cases}$$

The output is always sample-based, and the output port rate and data type (real/complex) are the same as those of the input port.

For convenience, length-M 1-D vector inputs and *sample-based* length-M row vector inputs are processed as single channels (i.e., as M-by-1 column vectors), and the output has the same dimension as the input.

Dialog Box



See Also

Complex Cepstrum	DSP Blockset
FFT	DSP Blockset
IDCT	DSP Blockset
Real Cepstrum	DSP Blockset
`dct`	Signal Processing Toolbox

Purpose

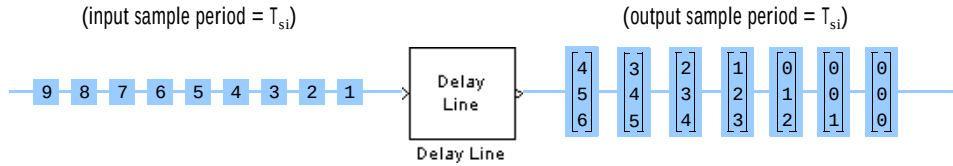
Rebuffer a sequence of inputs with a one-sample shift.

Library

Signal Management / Buffers

Description

The Delay Line block buffers the input samples into a sequence of overlapping or underlapping matrix outputs. In the most typical use (sample-based inputs), each output differs from the preceding output by only one sample, as illustrated below for scalar input.



Note that the first output of the block in the example above is all zeros; this is because the **Initial Conditions** parameter is set to zero. Due to the latency of the Delay Line block, all outputs are delayed by one frame, the entries of which are defined by the **Initial Conditions** parameter.

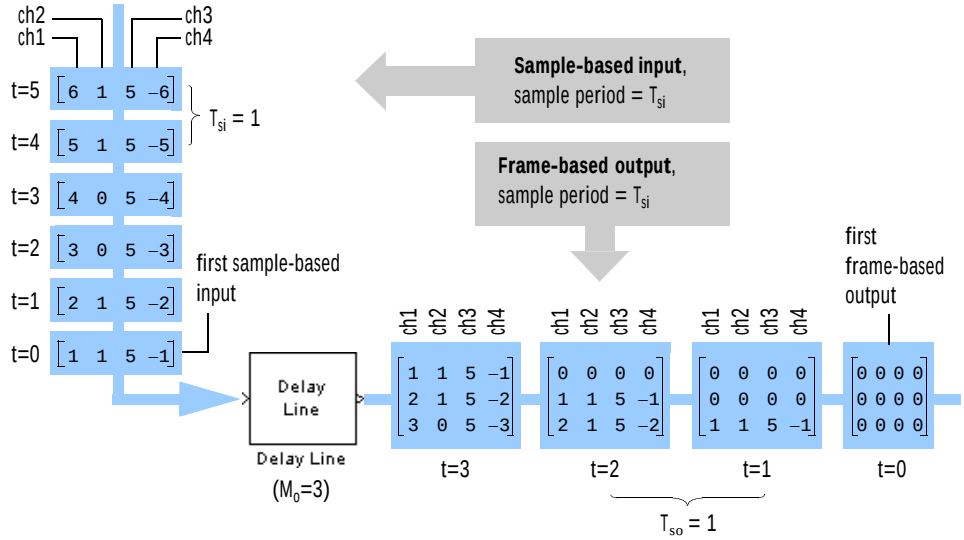
Sample-Based Operation

In sample-based operation, the Delay Line block buffers a sequence of sample-based length-N vector inputs (1-D, row, or column) into a sequence of overlapping frame-based M_o-by-N matrix outputs, where M_o is specified by the **Delay line size** parameter (M_o>1). That is, each input vector becomes a *row* in the frame-based output matrix.

At each sample time the new input vector is added in the last row of the output, so each output overlaps the previous output by M_o-1 samples. Therefore, the output sample period and frame period is the same as the input sample period ($T_{so}=T_{si}$, and $T_{fo}=T_{si}$). When M_o=1, the input is simply passed through to the output and retains the same dimension, but becomes frame-based. The latency of the block always causes an initial delay in the output; the value of the first output is specified by the **Initial conditions** parameter (see "Initial Conditions" below). Sample-based full-dimension matrix inputs are not accepted.

The Delay Line block's sample-based operation is similar to that of a Buffer block with **Buffer size** equal to M_o and **Buffer overlap** equal to M_o-1, except that the Buffer block has a different latency.

In the model below, the block operates on a sample-based input with a **Delay line size** of 3.



The input vectors in the example above do not begin appearing at the output until the second row of the second matrix due to the block's latency (see "Initial Conditions" below). The first output matrix (all zeros in this example) reflects the block's **Initial conditions** setting. As for any sample-based input, the output frame rate and output sample rate are both equal to the input sample rate.

Frame-Based Operation

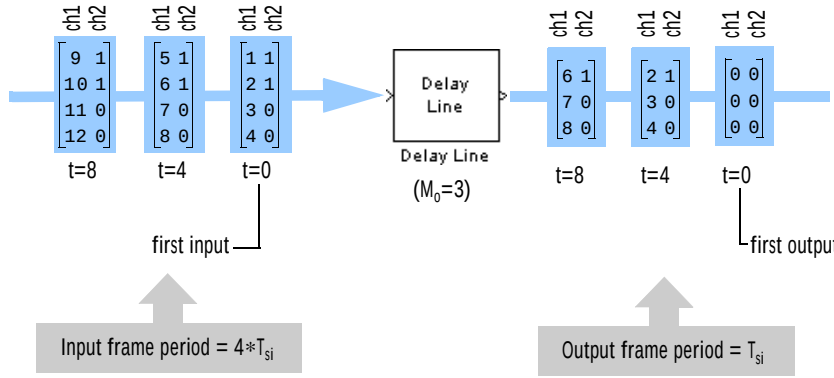
In frame-based operation, the Delay Line block rebuffers a sequence of frame-based M_i-by-N matrix inputs into a sequence of frame-based M_o-by-N matrix outputs, where M_o is the output frame size specified by the **Delay line size** parameter. Depending on whether M_o is greater than, less than, or equal to the input frame size, M_i, the output frames can be underlapped or overlapped. Each of the N input channels is rebuffered independently.

When $M_o > M_i$, the output frame overlap is the difference between the output and input frame size, M_o-M_i. When $M_o < M_i$, the output is underlapped; the Delay Line block discards the first M_i-M_o samples of each input frame so that only the last M_o samples are buffered into the corresponding output frame.

When $M_o = M_i$, the output data is identical to the input data, but is delayed by the latency of the block. Due to the block's latency, the outputs are always delayed by one frame, the entries of which are specified by the **Initial conditions** (see "Initial Conditions" below).

The output frame period is equal to the input frame period ($T_{fo}=T_{fi}$). The output sample period, T_{so}, is therefore equal to T_{fi}/M_o, or equivalently, $T_{si}(M_i/M_o)$

In the model below, the block rebuffers a two-channel frame-based input with a **Delay line size** of 3.



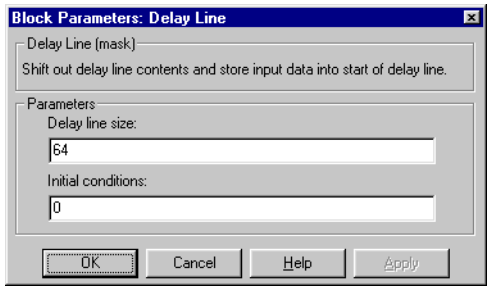
The first output frame in the example is a product of the latency of the Delay Line block; it is all zeros because the **Initial conditions** is set to be zero. Since the input frame size, 4, is larger than the output frame size, 3, only the last three samples in each input frame are propagated to the corresponding output frame. The frame periods of the input and output are the same, and the output sample period is $T_{si}(M_i/M_o)$, or 4/3 the input sample period.

Initial Conditions

The Delay Line block's buffer is initialized to the value specified by the **Initial condition** parameter. The block outputs this buffer at the first simulation step (t=0). If the block's output is a vector, the **Initial condition** can be a vector of the same size, or a scalar value to be repeated across all elements of the initial output. If the block's output is a matrix, the **Initial condition** can be a matrix of the same size, a vector (of length equal to the number of matrix rows) to be repeated across all columns of the initial output, or a scalar to be repeated across all elements of the initial output.

Delay Line

Dialog Box



Delay line size

The number of rows in output matrix, M_o.

Initial conditions

The value of the block's initial output, a scalar, vector, or matrix.

See Also

Buffer	DSP Blockset
Triggered Delay Line	DSP Blockset

See "Buffering Sample-Based and Frame-Based Signals" on page 3-47 for related information.

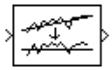
Purpose

Remove a linear trend from a vector.

Library

Statistics

Description



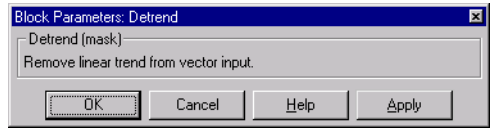
The Detrend block removes a linear trend from the length-M input vector, u, by subtracting the straight line that best fits the data in the least-squares sense.

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



See Also

Cumulative Sum	DSP Blockset
Difference	DSP Blockset
Least Squares Polynomial Fit	DSP Blockset
Unwrap	DSP Blockset
`detrend`	MATLAB

Difference

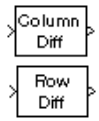
Purpose

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.

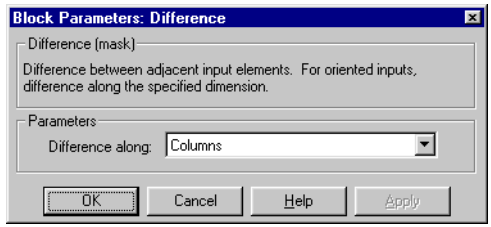
```
y = diff(u,[],2)       % Equivalent MATLAB code
```

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



Difference along

The dimension along which to compute element-to-element differences. **Columns** specifies columnwise differencing, while **Rows** specifies rowwise differencing.

See Also

Cumulative Sum — DSP Blockset

`diff` — MATLAB

Digital FIR Filter Design

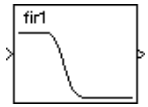
Purpose

Design and implement a variety of FIR filters.

Library

Filtering / Filter Designs

Description



The Digital FIR Filter Design block designs a discrete-time (digital) FIR filter in one of several different band configurations using a window method. Most of these filters are designed using the `fir1` function in the Signal Processing Toolbox, and are real with linear phase response. The block applies the filter to a discrete-time input using the Direct-Form II Transpose Filter block.

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

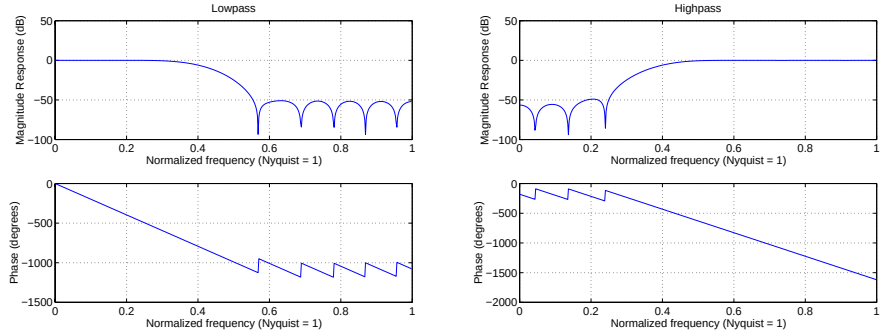
For complete details on the classical FIR filter design algorithm, see the description of the `fir1` and `fir2` functions in the Signal Processing Toolbox documentation.

Band Configurations

The band configuration for the filter is set from the **Filter type** pop-up menu. The band configuration parameters below this pop-up menu adapt appropriately to match the **Filter type** selection.

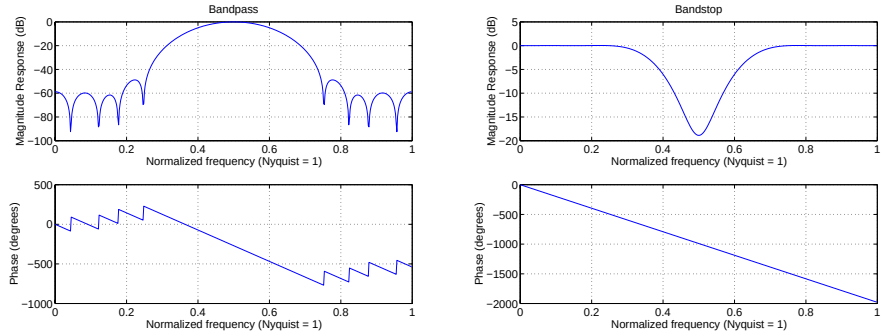
- **Lowpass** and **Highpass**

  In lowpass and highpass configurations, the **Filter order** and **Cutoff frequency** parameters specify the filter design. Frequencies are normalized to half the sample frequency. The figure below shows the frequency response of the default order-22 filter with cutoff at 0.4.



- **Bandpass** and **Bandstop**

  In bandpass and bandstop configurations, the **Filter order**, **Lower cutoff frequency**, and **Upper cutoff frequency** parameters specify the filter design. Frequencies are normalized to half the sample frequency, and the actual filter order is twice the **Filter order** parameter value. The figure below shows the frequency response of the default order-22 filter with lower cutoff at 0.4, and upper cutoff at 0.6.

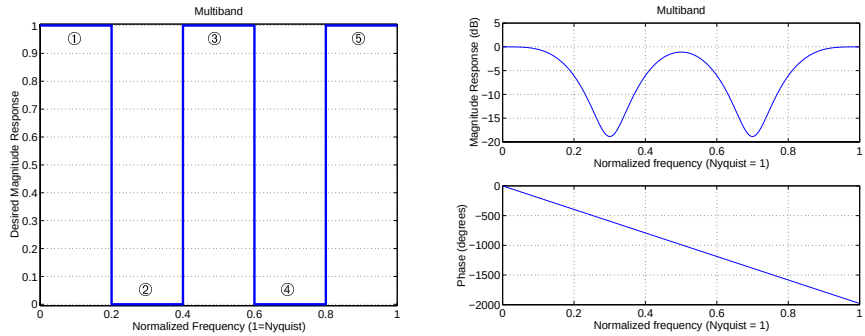


- **Multiband**

  In the multiband configuration, the **Filter order**, **Cutoff frequency vector**, and **Gain in the first band** parameters specify the filter design. The **Cutoff frequency vector** contains frequency points in the range `0` to `1`, where `1` corresponds to half the sample frequency. Frequency points must appear in ascending order. The **Gain in the first band** parameter specifies the gain in the first band: `0` indicates a stopband, and `1` indicates a passband. Additional

bands alternate between passband and stopband. The figure below shows the frequency response of the default order-22 filter with five bands, the first a passband.

Cutoff frequency vector = `[0.2 0.4 0.6 0.8]`



- **Arbitrary shape**

  In the arbitrary shape configuration, the **Filter order**, **Frequency vector**, and **Gains at these frequencies** parameters specify the filter design. The **Frequency vector**, `fn`, contains frequency points in the range `0` to `1` (inclusive) in ascending order, where `1` corresponds to half the sample frequency. The **Gains at these frequencies** parameter, `mn`, is a vector containing the desired magnitude response at the corresponding points in the **Frequency vector**. (Note that the specifications for the **Arbitrary shape** configuration are similar to those for the Yule-Walker IIR Filter Design block. Arbitrary-shape filters are designed using the `fir2` function in the Signal Processing Toolbox.)

  The desired magnitude response of the design can be displayed by typing

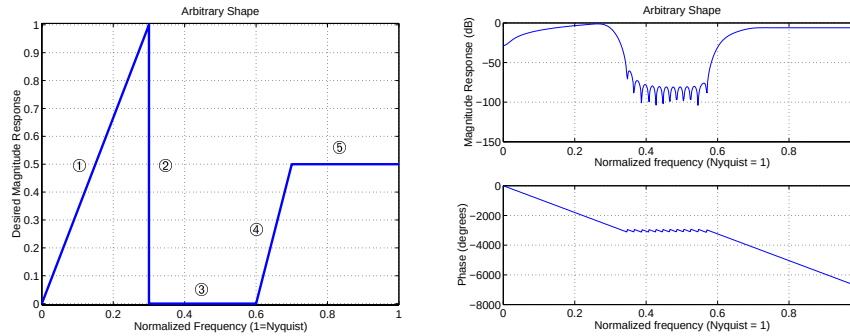
  ```
  plot(fn,mn)
  ```

  Duplicate frequencies can be used to specify a step in the response (such as band 2 below). The figure shows an order-100 filter with five bands.

Frequency = `[0.0 0.3 0.3 0.6 0.7 1.0]`

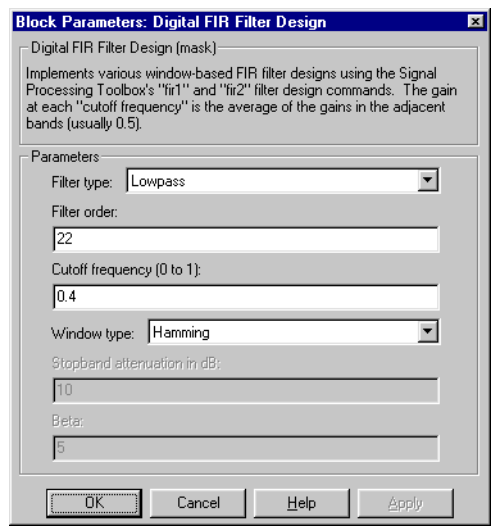
Gains = `[0.0 1.0 0.0 0.0 0.5 0.5]`

Band: ① ② ③ ④ ⑤



The **Window type** parameter allows you to select from a variety of different windows. See the Window Function block reference for a complete description of the available options.

Dialog Box



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

Filter type

The type of filter to design: **Lowpass**, **Highpass**, **Bandpass**, **Bandstop**, **Multiband**, or **Arbitrary Shape**. Tunable.

Filter order

The order of the filter. The filter length is one more than this value. For the **Bandpass** and **Bandstop** configurations, the order of the final filter is twice this value.

Cutoff frequency

The normalized cutoff frequency for the **Highpass** and **Lowpass** filter configurations. A value of `1` specifies half the sample frequency. Tunable.

Lower cutoff frequency

The lower passband or stopband frequency for the **Bandpass** and **Bandstop** filter configurations. A value of `1` specifies half the sample frequency. Tunable.

Upper cutoff frequency

The upper passband or stopband frequency for the **Bandpass** and **Bandstop** filter configurations. A value of `1` specifies half the sample frequency. Tunable.

Cutoff frequency vector

A vector of ascending frequency points defining the cutoff edges for the **Multiband** filter. A value of `1` specifies half the sample frequency. Tunable.

Gain in the first band

The gain in the first band of the **Multiband** filter: `0` specifies a stopband, `1` specifies a passband. Additional bands alternate between passband and stopband. Tunable.

Frequency vector

A vector of ascending frequency points defining the frequency bands of the **Arbitrary shape** filter. The frequency range is `0` to `1` including the endpoints, where `1` corresponds to half the sample frequency. Tunable.

Gains at these frequencies

A vector containing the desired magnitude response for the **Arbitrary shape** filter at the corresponding points in the **Frequency vector**. Tunable.

Window type

The type of window to apply. See the Window Function block reference. Tunable.

Stopband ripple

The level (dB) of stopband ripple, R_S, for the **Chebyshev** window. Tunable.

Beta

The **Kaiser** window β parameter. Increasing **Beta** widens the mainlobe and decreases the amplitude of the window sidelobes in the window's frequency magnitude response. Tunable.

References

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

Oppenheim, A. V. and R. W. Schafer. *Discrete-Time Signal Processing.* Englewood Cliffs, NJ: Prentice Hall, 1989.

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

See Also

Digital IIR Filter Design	DSP Blockset
Least Squares FIR Filter Design	DSP Blockset
Digital FIR Raised Cosine Filter Design	DSP Blockset
Remez FIR Filter Design	DSP Blockset
Window Function	DSP Blockset
Yule-Walker IIR Filter Design	DSP Blockset
`fir1`	Signal Processing Toolbox
`fir2`	Signal Processing Toolbox

See "Filter Designs" on page 4-3 for related information.

Digital FIR Raised Cosine Filter Design

Purpose

Design and implement a raised cosine FIR filter.

Library

Filtering / Filter Designs

Description

The Digital FIR Raised Cosine Filter Design block uses the `firrcos` function in the Signal Processing Toolbox to design a lowpass, linear-phase, digital FIR filter with a raised cosine transition band. The block applies the filter to a discrete-time input using the Direct-Form II Transpose Filter block.

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

The frequency response of the raised cosine filter is

$$H(f) = \begin{cases} \dfrac{1}{2f_{n0}} & 0 \le |f| \le (1-R)f_{n0} \\ \left[\dfrac{1+\cos\dfrac{\pi}{2Rf_{n0}}|f|-(1-R)f_{n0}}{4f_{n0}}\right] & (1-R)f_{n0} \le |f| \le (1+R)f_{n0} \\ 0 & (1+R)f_{n0} \le |f| \le 1 \end{cases}$$

where $H(f)$ is the magnitude response at frequency f, f_{n0} is the normalized cutoff frequency (-6 dB) specified by the **Upper cutoff frequency** parameter, and R is a rolloff factor in the range `[0,1]` determining the passband-to-stopband transition width.

The **Square-root raised cosine filter** option designs a filter with magnitude response $\sqrt{H(f)}$. This is useful when the filter is part of a pair of matched filters.

When the **Design method** parameter is set to **Rolloff factor**, the secondary **Rolloff factor** parameter is enabled, and R can be directly specified. When **Design method** is set to **Transition bandwidth**, the secondary **Transition bandwidth** parameter is enabled, and the transition region bandwidth, Δf, can

be specified in place of R. The transition region is centered on f_{n0} and must be sufficiently narrow to satisfy

$$0 < f_{n0} \pm \frac{\Delta f}{2} < 1$$

The **Upper cutoff frequency** and **Transition bandwidth** parameter values are normalized to half the sample frequency.

The **Window type** parameter allows you to apply a variety of different windows to the raised cosine filter. See the Window Function block reference for a complete description of the available options.

Algorithm

The filter output is computed by convolving the input with a truncated, delayed, windowed version of the filter's impulse response. The impulse response for the raised cosine filter is

$$h(kT_s) = \frac{1}{F_s}\operatorname{sinc}(2kT_s f_{n0})\frac{\cos(2\pi RkT_s f_{n0})}{1-(4RkT_s f_{n0})^2} \qquad -\infty < k < \infty$$

which has limits

$$h(0) = \frac{1}{F_s}$$

and

$$h\left(\pm\frac{1}{4Rf_{n0}}\right) = \frac{R}{2F_s}\sin\left(\frac{\pi}{2R}\right)$$

The impulse response for the square-root raised cosine filter is

$$h(kT_s) = \frac{4R\cos((1+R)2\pi kT_s f_{n0}) + \dfrac{\sin((1-R)2\pi kT_s f_{n0})}{8RkT_s f_{n0}}}{\pi F_s\sqrt{\dfrac{1}{2f_{n0}}}((8RkT_s f_{n0})^2 - 1)} \qquad -\infty < k < \infty$$

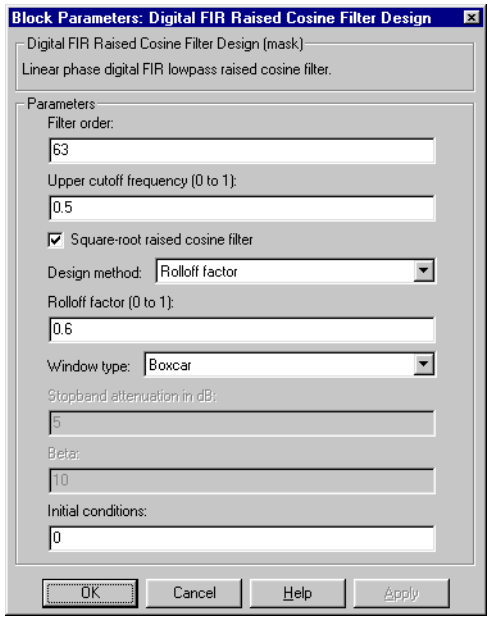
which has limits

$$h(0) = (-4R - \pi + \pi R)\frac{\sqrt{2f_{n0}}}{\pi F_s}$$

and

$$h\left(\pm\frac{1}{8Rf_{n0}}\right) = -\frac{\sqrt{2f_{n0}}}{2\pi F_s}\left[\pi(1+R)\sin\frac{\pi(1+R)}{4R} - 4R\sin\frac{\pi(R-1)}{4R} + \pi(R-1)\cos\frac{\pi(R-1)}{4R}\right]$$

Dialog Box



Filter order

The order of the filter. The filter length is one more than this value.

Upper cutoff frequency

The normalized cutoff frequency, f_{n0}. A value of `1` specifies half the sample frequency. Tunable.

Square-root raised cosine filter

Selects the square-root filter option, which designs a filter with magnitude response $\sqrt{H(f)}$. Tunable.

Design method

The method used to design the transition region of the filter, **Rolloff factor** or **Transition bandwidth**. Tunable.

Rolloff factor

The rolloff factor, *R*, enabled when **Rolloff factor** is selected in the **Design method** parameter. Tunable.

Transition bandwidth

The transition bandwidth, Δf, enabled when **Transition bandwidth** is selected in the **Design method** parameter. Tunable.

Window type

The type of window to apply. See the Window Function block reference. Tunable.

Stopband attenuation in dB

The level (dB) of stopband attenuation, R_s, for the **Chebyshev** window. Tunable.

Beta

The **Kaiser** window β parameter. Increasing β widens the mainlobe and decreases the amplitude of the window sidelobes in the window's frequency magnitude response. Tunable.

Initial conditions

The filter's initial conditions, a scalar, vector, or matrix. See the Direct-Form II Transpose Filter block reference for complete syntax information.

References

Proakis, J. G. *Digital Communications*. Third ed. New York, NY: McGraw-Hill, 1995.

Proakis, J. G. and M. Salehi. *Contemporary Communication Systems Using MATLAB*. Boston, MA: PWS Publishing, 1998.

Digital FIR Raised Cosine Filter Design

See Also

Digital FIR Filter Design	DSP Blockset
Digital IIR Filter Design	DSP Blockset
Direct-Form II Transpose Filter	DSP Blockset
Least Squares FIR Filter Design	DSP Blockset
Remez FIR Filter Design	DSP Blockset
Window Function	DSP Blockset
Yule-Walker IIR Filter Design	DSP Blockset
`firrcos`	Signal Processing Toolbox

See “Filter Designs” on page 4-3 for related information.

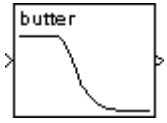
Purpose

Design and implement an IIR filter.

Library

Filtering / Filter Designs

Description



The Digital IIR Filter Design block designs a discrete-time (digital) IIR filter in a lowpass, highpass, bandpass, or bandstop configuration, and applies it to the input using the Direct-Form II Transpose Filter block.

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

The **Design method** parameter allows you to specify Butterworth, Chebyshev type I, Chebyshev type II, and elliptic filter designs. Note that for the bandpass and bandstop configurations, the actual filter length is twice the **Filter order** parameter value.

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

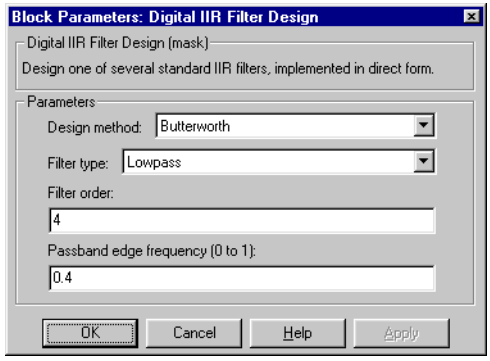
The design and band configuration of the filter are selected from the **Design method** and **Filter type** pop-up menus in the dialog box. For each combination of design method and band configuration, an appropriate set of secondary parameters is displayed.

The table below lists the available parameters for each design/band combination. For lowpass and highpass band configurations, these parameters include the passband edge frequency f_{np}, the stopband edge frequency f_{ns}, the passband ripple R_p, and the stopband attenuation R_s. For bandpass and bandstop configurations, the parameters include the lower and upper passband edge frequencies, f_{np1} and f_{np2}, the lower and upper stopband edge frequencies, f_{ns1} and f_{ns2}, the passband ripple R_p, and the stopband attenuation R_s. Frequency values are normalized to half the sample frequency, and ripple and attenuation values are in dB.

	Lowpass	Highpass	Bandpass	Bandstop
Butterworth	Order, f_{np}	Order, f_{np}	Order, f_{np1}, f_{np2}	Order, f_{np1}, f_{np2}
Chebyshev Type I	Order, f_{np}, R_p	Order, f_{np}, R_p	Order, f_{np1}, f_{np2}, R_p	Order, f_{np1}, f_{np2}, R_p
Chebyshev Type II	Order, f_{ns}, R_s	Order, f_{ns}, R_s	Order, f_{ns1}, f_{ns2}, R_s	Order, f_{ns1}, f_{ns2}, R_s
Elliptic	Order, f_{np}, R_p, R_s	Order, f_{np}, R_p, R_s	Order, f_{np1}, f_{np2}, R_p, R_s	Order, f_{np1}, f_{np2}, R_p, R_s

The digital filters are designed using the Signal Processing Toolbox's filter design commands `butter`, `cheby1`, `cheby2`, and `ellip`.

Dialog Box



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

Design method

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

Filter type

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

Filter order

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

Passband edge frequency

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

Lower passband edge frequency

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

Upper passband edge frequency

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

Stopband edge frequency

The normalized stopband edge frequency for the highpass and lowpass band configurations of the Chebyshev type II design. Tunable.

Lower stopband edge frequency

The normalized lower stopband frequency for the bandpass and bandstop configurations of the Chebyshev type II design. Tunable.

Upper stopband edge frequency

The normalized upper stopband frequency for the bandpass and bandstop filter configurations of the Chebyshev type II design. Tunable.

Passband ripple in dB

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

Stopband attenuation in dB

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

References

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

Oppenheim, A. V. and R. W. Schafer. *Discrete-Time Signal Processing.* Englewood Cliffs, NJ: Prentice Hall, 1989.

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

See Also

Analog Filter Design	DSP Blockset
Digital FIR Filter Design	DSP Blockset
Yule-Walker IIR Filter Design	DSP Blockset
`butter`	Signal Processing Toolbox
`cheby1`	Signal Processing Toolbox
`cheby2`	Signal Processing Toolbox
`ellip`	Signal Processing Toolbox

See “Filter Designs” on page 4-3 for related information.

Purpose

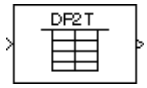
Apply an IIR filter to the input.

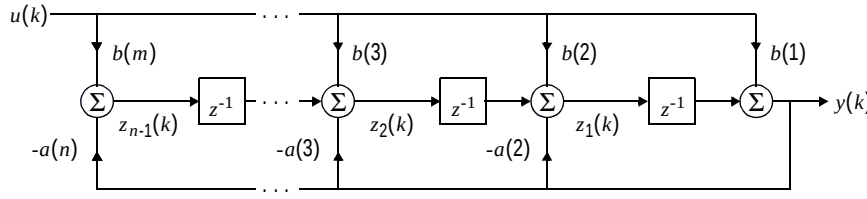
Library

Filtering / Filter Structures

Description

The Direct-Form II Transpose Filter block applies a transposed direct-form II IIR filter to the input.





This is a canonical form that has the minimum number of delay elements. The filter order is `max(m,n)-1`.

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

The filter is specified in the parameter dialog box by its transfer function,

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_1 + b_2 z^{-1} + \ldots + b_{m+1} z^{-(m-1)}}{a_1 + a_2 z^{-1} + \ldots + a_{n+1} z^{-(n-1)}}$$

where the **Numerator** parameter specifies the vector of numerator coefficients,

```
[b(1) b(2) ... b(m)]
```

and the **Denominator** parameter specifies the vector of denominator coefficients,

```
[a(1) a(2) ... a(n)]
```

The filter coefficients are normalized by a_1.

Initial Conditions

In its default form, the filter initializes the internal filter states to zero, which is equivalent to assuming past inputs and outputs are zero. The block also accepts optional nonzero initial conditions for the filter delays. Note that the number of filter states (delay elements) per input channel is

```
max(m,n)-1
```

The **Initial conditions** parameter may take one of four forms:

- Empty matrix

  The empty matrix, `[]`, causes a zero (0) initial condition to be applied to all delay elements in each filter channel.

- Scalar

  The scalar value is copied to all delay elements in each filter channel. Note that a value of zero is equivalent to setting the **Initial conditions** parameter to the empty matrix, `[]`.

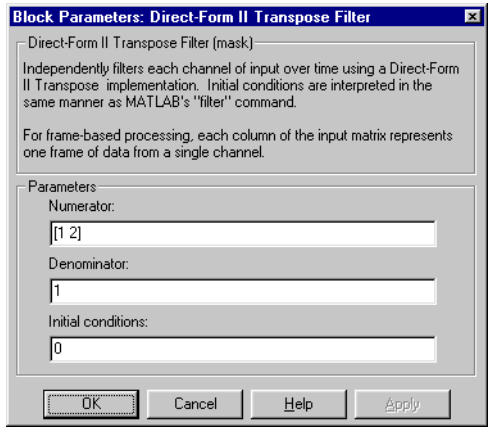
- Vector

  The vector has a length equal to the number of delay elements in each filter channel, `max(m,n)-1`, and specifies a unique initial condition for each delay element in the filter channel. This vector of initial conditions is applied to each filter channel.

- Matrix

  The matrix specifies a unique initial condition for each delay element, and can specify different initial conditions for each filter channel. The matrix must have the same number of rows as the number of delay elements in the filter, `max(m,n)-1`, and must have one column per filter channel.

Dialog Box



Numerator

The filter numerator vector. Tunable; the numerator coefficients can be adjusted while the simulation runs, but the vector length (i.e., the filter order) must remain the same.

Denominator

The filter denominator vector. Tunable; the denominator coefficients can be adjusted while the simulation runs, but the vector length (i.e., the filter order) must remain the same.

Initial conditions

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

References

Oppenheim, A. V. and R. W. Schafer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989.

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

See Also

Biquadratic Filter	DSP Blockset
Discrete Filter	Simulink
Filter Realization Wizard	DSP Blockset
Time-Varying Direct-Form II Transpose Filter	DSP Blockset
`filter`	MATLAB

See “Filter Structures” on page 4-23 for related information.

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.

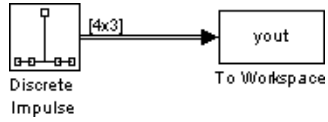
When `D` is a length-N vector, the block generates an M-by-N matrix output representing N distinct channels, where frame size M is specified by the **Samples per frame** parameter. The impulse for the `i`th channel appears at sample `D(i)+1`. For M=1, the output is sample-based; otherwise, the output is frame-based.

The **Sample time** parameter value, T_s, specifies the output signal sample period. The resulting frame period is $M*T_s$.

The **Data type** parameter allows you to specify an output precision of double, single, or Boolean. Note, however, 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.

Example

Construct the model below.



Configure the Discrete Impulse block to generate a frame-based three-channel output of type double, with impulses at samples 1, 4, and 6 of channels 1, 2, and 3, respectively. Use a sample period of 0.25 and a frame size of 4. The corresponding settings should be as follows:

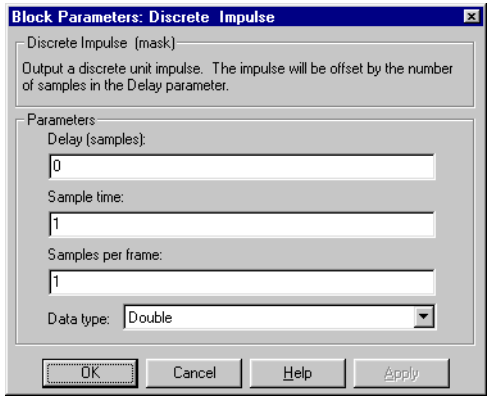
- **Delay** = `[0 3 5]`
- **Sample time** = `0.25`
- **Samples per frame** = `4`
- **Data type** = **Double**

Run the model and look at the output, `yout`. The first few samples of each channel are shown below.

```
yout(1:10,:)
ans =

     1     0     0
     0     0     0
     0     0     0
     0     1     0
     0     0     0
     0     0     1
     0     0     0
     0     0     0
     0     0     0
     0     0     0
```

The block generates an impulse at sample 1 of channel 1 (first column), at sample 4 of channel 2 (second column), and at sample 6 of channel 3 (third column).

Dialog Box



Delay

The number of zero-valued output samples, `D`, preceding the impulse. A length-N vector specifies an N-channel output.

Sample time

The sample period, T_s, of the output signal. The output frame period is $M*T_s$.

Samples per frame

The number of samples, M, in each output frame.

Data type

The precision of the output.

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

See “Creating Signals Using Signal Generator Blocks” on page 3-36 for related information.

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 $0 \le D < K$, 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.

Sample-Based Operation

When the input is sample-based, the block treats each of the M∗N matrix elements as an independent channel, and downsamples each channel over time. The input and output sizes are identical.

The **Sample-based mode** parameter determines how the block represents the new rate at the output. There are two available options:

- **Allow multirate**

  When **Allow multirate** is selected, the sample period of the sample-based output is K times longer than the input sample period ($T_{so} = KT_{si}$). The block is therefore multirate.

- **Enforce single rate**

  When **Enforce single rate** is selected, the block forces the output sample rate to match the input sample rate ($T_{so} = T_{si}$) by repeating every Kth input sample K times at the output. The block is therefore single-rate. (The block's operation when **Enforce single rate** is selected is similar to the operation of a Sample and Hold block with a repeating trigger event of period KT_{si}.)

The setting of the **Frame-based mode** popup menu does not affect sample-based inputs.

Frame-Based Inputs

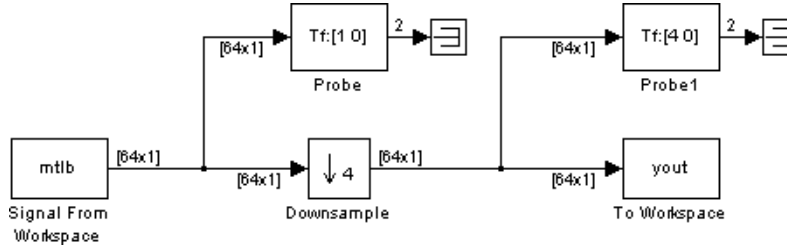
When the input is frame-based, the block treats each of the N input columns as a frame containing M_i sequential time samples from an independent channel. The block downsamples each channel independently by discarding K-1 rows of the input matrix following each row that it passes through to the output. The downsample factor must be less than the frame size, $K < M_i$.

The **Frame-based mode** parameter determines how the block adjusts the rate at the output to accommodate the reduced number of samples. There are two available options:

- **Maintain input frame size**

  The block generates the output at the slower (downsampled) rate by using a proportionally longer frame *period* at the output port than at the input port. For downsampling by a factor of K, the output frame period is K times longer than the input frame period ($T_{fo} = KT_{fi}$), but the input and output frame sizes are equal.

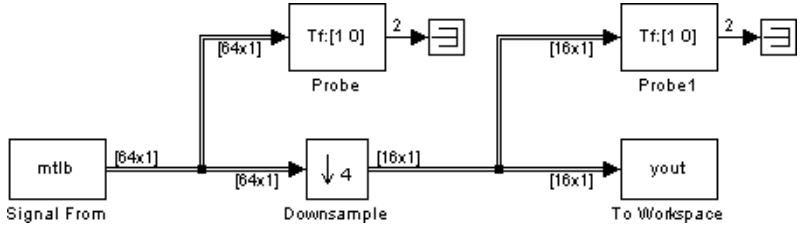
  The model below shows a single-channel input with a frame period of 1 second being downsampled by a factor of 4 to a frame period of 4 seconds. The input and output frame sizes are identical.



- **Maintain input frame rate**

  The block generates the output at the slower (downsampled) rate by using a proportionally smaller frame *size* than the input. For downsampling by a factor of K, the output frame size is K times smaller than the input frame size ($M_o = M_i/K$), but the input and output frame rates are equal.

  The model below shows a single-channel input of frame size 64 being downsampled by a factor of 4 to a frame size of 16. The input and output frame rates are identical.



The setting of the **Sample-based mode** popup menu does not affect frame-based inputs.

Latency

Zero Latency. The Downsample block has *zero tasking latency* for the special combinations of input signal sampling and parameter settings shown in the table below. In all of these cases the block has single-rate operation.

Input Sampling	Parameter Settings
Sample-based	**Downsample factor** parameter, K, is `1`, *or* **Enforce single rate** is selected (with D=0)
Frame-based	**Downsample factor** parameter, K, is `1`, *or* **Maintain input frame rate** is selected

Zero tasking latency means that the block propagates input sample D+1 (received at *t*=0) as the first output sample, followed by input sample D+1+K, input sample D+1+2K, and so on. The **Initial condition** parameter value is not used.

Nonzero Latency. The Downsample block is multirate for most settings other than those in the above table. The amount of latency for multirate operation depends on input signal sampling and Simulink's tasking mode, as shown in the table below.

Multirate...	Sample-Based Latency	Frame-Based Latency
Single-tasking	None, for D=0 One sample, for D>0	One frame (M_i samples)
Multitasking	One sample	One frame (M_i samples)

The only case of nonzero single-rate latency occurs in sample-based mode, when **Enforce single rate** is selected with D > 0. The latency in this case is one sample.

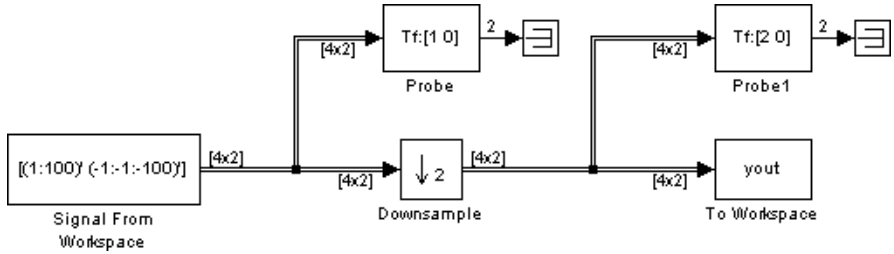
In all cases of *one-sample latency*, the initial condition for each channel appears as the first output sample. Input sample D+1 appears as the second output sample for each channel, followed by input sample D+1+K, input sample D+1+2K, and so on. The **Initial condition** parameter can be an M_i-by-N matrix containing one value for each channel, or a scalar to be applied to all signal channels.

In all cases of *one-frame latency*, the M_i rows of the initial condition matrix appear in sequence as the first M_i output rows. Input sample D+1 (i.e, row D+1 of the input matrix) appears in the output as sample M_i+1, followed by input sample D+1+K, input sample D+1+2K, and so on. The **Initial condition** value can be an M_i-by-N matrix, or a scalar to be repeated across all elements of the M_i-by-N matrix. See the example below for an illustration of this case.

See "Excess Algorithmic Delay (Tasking Latency)" on page 3-91 and "The Simulation Parameters Dialog Box" in the Simulink documentation for more information about block rates and Simulink's 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 seconds. 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 deselecting 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 Simulink's multitasking mode, select **Fixed-step** and **discrete** from the **Type** controls in the **Solver**

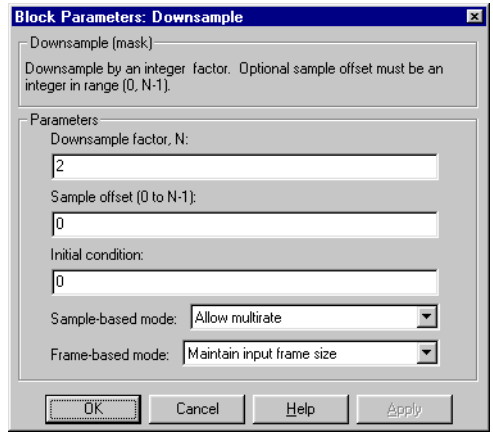
panel of the **Simulation Parameters** dialog box, and select **MultiTasking** from the **Mode** parameter. Additionally, set the **Stop time** to `30`.

Run the model and look at the output, `yout`. The first few samples of each channel are shown below.

```
yout =

    11   -11
    12   -12
    13   -13
    14   -14
     2    -2
     4    -4
     6    -6
     8    -8
    10   -10
    12   -12
    14   -14
```

Since we ran this frame-based multirate model in multitasking mode, the first row of the initial condition matrix appears as the first output sample, followed by the other three initial condition rows. The second row of the first input matrix (i.e., row D+1, where D is the **Sample offset**) appears in the output as sample 5 (i.e., sample M_i+1, where M_i is the input frame size).

Dialog Box



Downsample factor

The integer factor, K, by which to decrease the input sample rate.

Sample offset

The sample offset, D, which must be an integer in the range [0, K-1].

Initial condition

The value with which the block is initialized for cases of nonzero latency; a scalar or matrix.

Sample-based mode

The method by which to implement downsampling for sample-based inputs: **Allow multirate** (i.e, decrease the output sample rate), or **Force single-rate** (i.e., force the output sample rate to match the input sample rate by repeating every Kth input sample K times at the output).

Frame-based mode

The method by which to implement downsampling for frame-based inputs: **Maintain input frame size** (i.e., decrease the frame rate), or **Maintain input frame rate** (i.e., decrease the frame size).

See Also

FIR Decimation	DSP Blockset
FIR Rate Conversion	DSP Blockset
Repeat	DSP Blockset
Sample and Hold	DSP Blockset
Upsample	DSP Blockset

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.

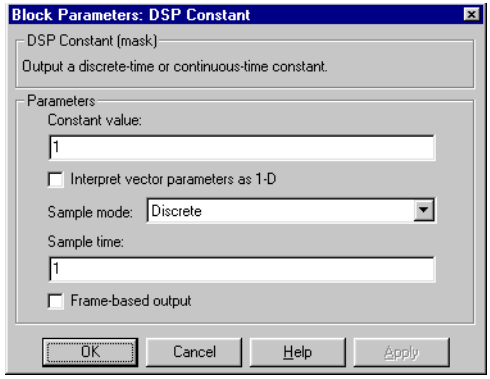
When a row vector or column vector is specified for the **Constant value** parameter, and the **Interpret vector parameters as 1-D** check box is selected, the dimension of the **Constant value** vector is disregarded, and the output is a 1-D vector. When the **Interpret vector parameters as 1-D** check box is *not* selected, the output dimension is constrained to match the **Constant value** dimension (row or column). When the **Constant value** is an M-by-N matrix with M > 1 and N > 1, the output is always M-by-N.

When **Sample mode** is set to **Continuous**, the output is a continuous-time signal. When **Sample mode** is set to **Discrete**, the **Sample time** and **Frame-based output** parameters are enabled, and signal has the discrete output period specified by the **Sample time** parameter.

When the **Frame-based output** check box is selected, the output is frame-based; otherwise, the output is sample-based. Because a 1-D vector signal cannot be frame-based, an active **Frame-based output** setting *overrides* an active **Interpret vector parameters as 1-D** setting, and the output is a frame-based column vector (regardless of the actual **Constant value** vector dimension).

DSP Constant

Dialog Box



Constant value

The constant to generate. Values entered here can be tuned, but their dimensions must remain fixed.

Interpret vector parameters as 1-D

When selected, generates 1-D vector outputs for **Constant value** settings with vector dimension; when unselected, generates 2-D vector outputs for **Constant value** settings with vector dimension. An active **Interpret vector parameters as 1-D** setting is *overridden* by an active **Frame-based output** setting.

Sample mode

The sample mode of the output, **Discrete** for a discrete-time signal or **Continuous** for a continuous-time signal.

Sample time

The discrete sample-period for sample-based outputs, or the discrete frame-period for frame-based outputs. This parameter is enabled when **Discrete** is selected in the **Sample mode** menu.

Frame-based output

Specifies frame-based output when selected. This parameter is enabled when **Discrete** is selected in the **Sample mode** menu

See Also

Constant
Signal From Workspace
Simulink
DSP Blockset

See “Creating Signals Using Constant Blocks” on page 3-33 for related information.

Dyadic Analysis Filter Bank

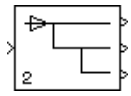
Purpose

Decompose a signal into components of equal or logarithmically decreasing frequency intervals and sample rates.

Library

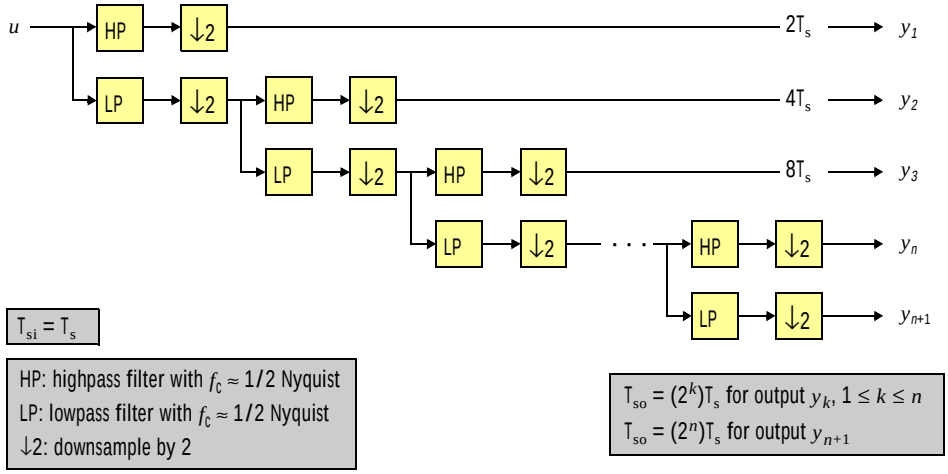
Filtering / Multirate Filters

Description



The Dyadic Analysis Filter Bank block decomposes a broadband signal into a collection of successively more bandlimited components by repeatedly dividing the frequency range. The typical (asymmetric) *n*-level filter bank structure is shown below.

Asymmetric Filter Bank, n Levels



At each level, the *low-frequency* output of the previous level is decomposed into adjacent high- and low-frequency subbands by a highpass (HP) and lowpass (LP) filter pair. Each of the two output subbands is half the bandwidth of the input to that level (hence "dyadic"). The bandlimited output of each filter is maximally decimated by a factor of 2 to preserve the bit rate of the original signal. In wavelet applications (see below) the aliasing introduced by the decimation stage can be exactly canceled in reconstruction.

The **Lowpass FIR filter coefficients** and **Highpass FIR filter coefficients** parameters specify (respectively) the filter coefficients to be used for every lowpass and highpass direct-form II transpose filter in the filter bank. The values of these coefficients are typically computed using the wavelet family

functions in the Wavelet Toolbox (see the Wavelet Toolbox documentation for more information).

Tree Structure

The **Tree structure** parameter specifies an asymmetric (or wavelet) tree, as shown above, or a symmetric structure, as shown below. Note that the symmetric structure decomposes both the high- and low-frequency subbands at each level, whereas the asymmetric structure only decomposes the low-frequency bands.

Asymmetric Tree. The asymmetric structure in the first figure (**Tree structure** set to **Asymmetric**) has n+1 outputs, where n is the **Number of levels** parameter value. The sample rate and bandwidth of the top output are half the input sample rate and bandwidth. The sample rate and bandwidth of each additional output (except the last) are half that of the output from the previous level. In general, for an input with sample period $T_{si} = T_s$, and bandwidth BW, output y_k has sample period $T_{so,k}$ and bandwidth BW_k.

$$T_{so,k} = \begin{cases} (2^k)T_s & (1 \le k \le n) \\ (2^n)T_s & (k = n+1) \end{cases}$$

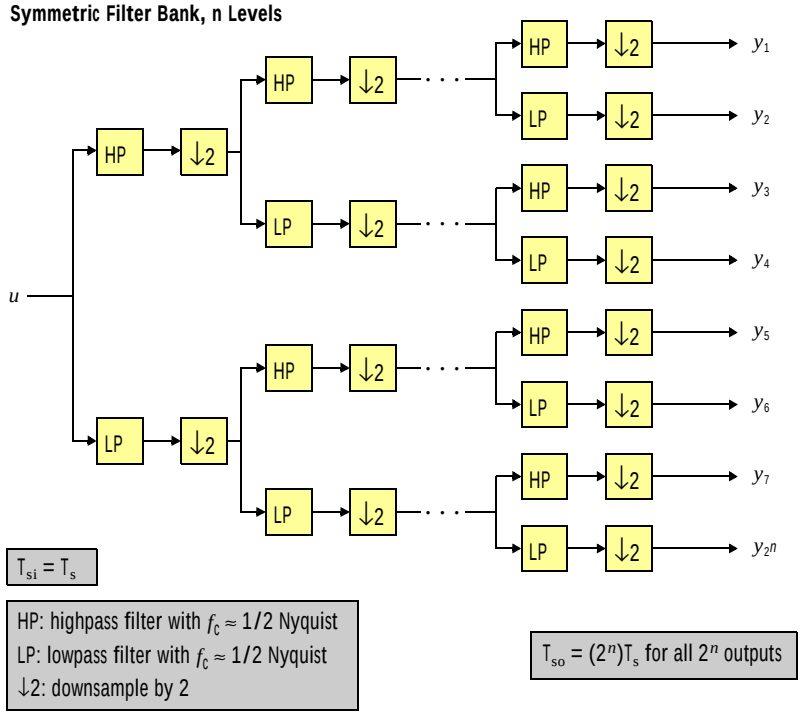
$$BW_k = \begin{cases} \dfrac{BW}{2^k} & (1 \le k \le n) \\ \dfrac{BW}{2^n} & (k = n+1) \end{cases}$$

Note that in frame-based mode, the change in the sample period of output y_k is reflected by its frame size, $M_{o,k}$, rather than by its frame rate.

$$M_{o,k} = \begin{cases} \dfrac{M_i}{2^k} & (1 \le k \le n) \\ \dfrac{M_i}{2^n} & (k = n+1) \end{cases}$$

The bottom two outputs (y_n and y_{n+1}) share the same sample period, bandwidth, and frame size because they originate at the same tree level.

Symmetric Tree. The symmetric structure shown below (**Tree structure** set to **Symmetric**) has 2^n outputs, where n is the **Number of levels** parameter value.



The sample rate and bandwidth of every output are reduced by a factor of 2^n from the input sample rate and bandwidth. For an input with sample period $T_{si} = T_s$, and bandwidth BW, output y_k has sample period $T_{so,k}$ and bandwidth BW_k.

$$T_{so,\,k} = (2^n)T_s \qquad (1 \le k \le 2^n)$$

$$BW_k = \frac{BW}{2^n} \qquad (1 \le k \le 2^n)$$

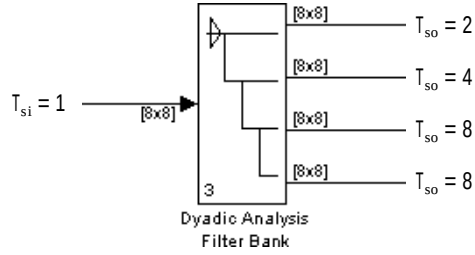
In frame-based mode, the sample period of output y_k is reflected by its frame size, $M_{o,k}$, rather than by its frame rate.

$$M_{o,\,k} = \frac{M_i}{2^n} \qquad (1 \le k \le 2^n)$$

Sample-Based Operation

An M-by-N sample-based matrix input is treated as M∗N independent channels, and the block filters each channel independently over time. The output at each port is the same size as the input, one output channel for each input channel. As described earlier, for the asymmetric tree structure, each output port has a different sample period.

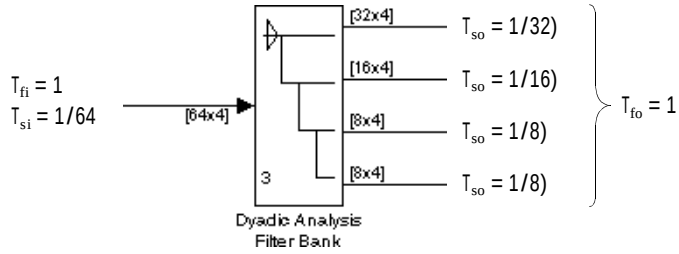
The figure below shows the input and output sample periods for a 64-channel sample-based input to a three-level filter bank. The input has a period of 1, so the fastest output has a period of 2.



Frame-Based Operation

An M_i-by-N frame-based matrix input is treated as N independent channels, and the block filters each channel independently over time. The input frame size M_i must be a multiple of 2^n, and n is the number of filter bank levels. For example, a frame size of 8 would be appropriate for a three-level tree (2^3=8). The number of columns in each output is the same as the number of columns in the input.

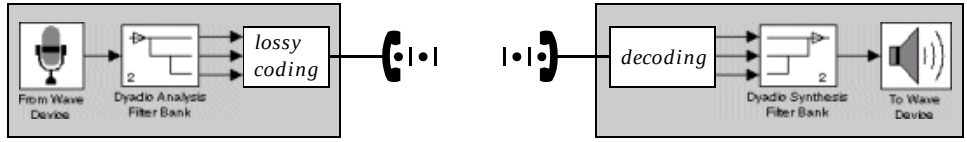
Each output port has the *same frame period* as the input. The reduction in the output sample rates results from the smaller output frame sizes, as shown in the example below for a four-channel input to a three-level asymmetric tree.



Applications

Wavelets. The primary application for dyadic analysis 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.

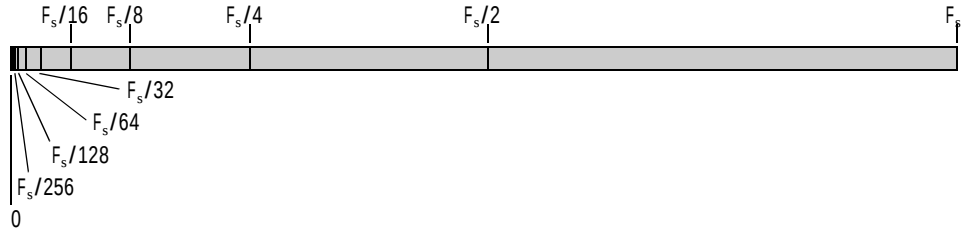


At the receiving end, the transmission is decoded and fed to a dyadic synthesis filter bank to reconstruct the original signal. The filter coefficients of the complementary analysis and synthesis stages are designed to cancel aliasing introduced by the filtering and resampling.

Scalograms. When the magnitudes in each of the subband signals y_k, $1 \le k \le n$, are plotted across the full bandwidth of the original signal, the result is a *scalogram*. This is the equivalent of a spectrogram with constant Q, where

$$Q = \frac{f_{y_k}}{BW_{y_k}}$$

and f_{y_k} is the midpoint frequency of the band occupied by output y_k. The frequency axis of a scalogram therefore has logarithmic divisions like those shown below, where F_s is the sample rate ($1/T_s$).



Latency

Zero Latency. The Dyadic Analysis Filter Bank block has *no tasking latency* for frame-based operation, which is always single-rate. The block therefore analyzes the first input sample (received at t=0) to produce the first output sample at each port.

Nonzero Latency. The Dyadic Analysis Filter Bank block has tasking latency only for sample-based operation, which is always multirate. As shown in the table below, the amount of latency, D, depends on the structure (symmetric or asymmetric) of the n-level tree used by the block.

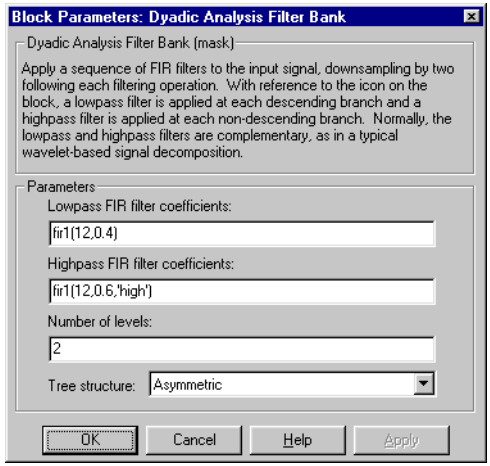
Multirate...	Symmetric Tree	Asymmetric Tree
Single-tasking	One sample	2^{n-1} samples
Multitasking	One sample	2^{n-1} samples

In the above cases, the block repeats a zero initial condition in each channel for the first D output samples. For example, in single-tasking mode, the asymmetric tree structure generates 2^{n-1} zero-valued output samples at each port in each channel before propagating the first analyzed input sample (computed from the input received at t=0).

See “Excess Algorithmic Delay (Tasking Latency)” in Chapter 3 and “The Simulation Parameters Dialog Box” in the Simulink documentation for more information about block rates and Simulink’s tasking modes.

Dyadic Analysis Filter Bank

Dialog Box



Lowpass FIR filter coefficients

A vector of filter coefficients (descending powers of *z*) to be shared by all the lowpass filters in the filter bank.

Highpass FIR filter coefficients

A vector of filter coefficients (descending powers of *z*) to be shared by all the highpass filters in the filter bank.

Number of levels

The number of filter bank levels. An *n*-level asymmetric structure has *n*+1 outputs; an *n*-level symmetric structure has 2^n outputs.

Tree structure

The structure of the filter bank, **Asymmetric** (wavelet) or **Symmetric**.

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.

See Also

Dyadic Synthesis Filter Bank	DSP Blockset
Wavelet Analysis	DSP Blockset

See the following sections for related information:

- “Converting Sample Rates and Frame Rates” on page 3-20
- “Multirate Filters” on page 4-24

Dyadic Synthesis Filter Bank

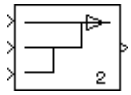
Purpose

Reconstruct a signal from its multirate bandlimited components.

Library

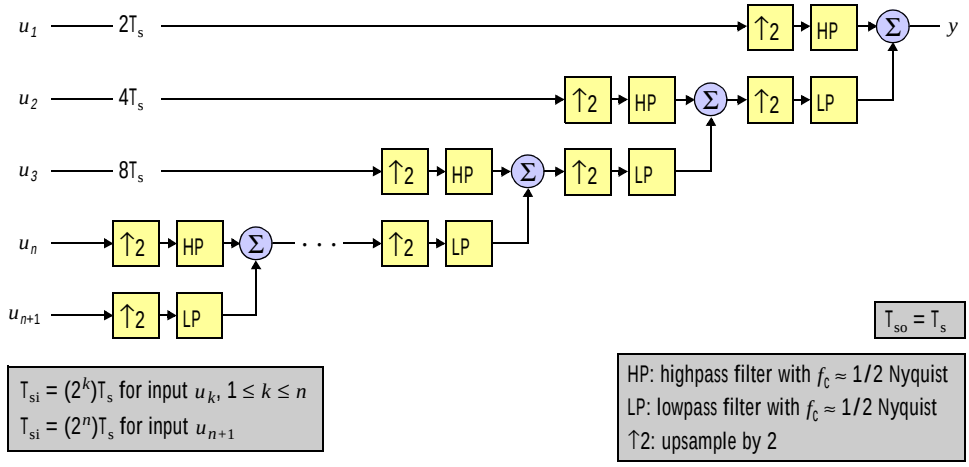
Filtering / Multirate Filters

Description



The Dyadic Synthesis Filter Bank block typically reconstructs a signal that was decomposed by the Dyadic Analysis Filter Bank block. The reconstruction or *synthesis* process is the inverse of the analysis process, and restores the original signal by upsampling, filtering, and summing the bandlimited inputs in stages corresponding to the analysis process. The typical (asymmetric) *n*-level filter bank structure is shown below.

Asymmetric Filter Bank, n Levels



At each level, the two bandlimited inputs (one low-frequency, one high-frequency, both with the same sample rate) are upsampled by a factor of 2 to match the sample rate of the input to the next stage. They are then filtered by a highpass (HP) and lowpass (LP) filter pair with coefficients calculated to cancel (in the subsequent summation) the aliasing introduced in the corresponding dyadic analysis filter stage. The output from each (upsample-filter-sum) level has twice the bandwidth and twice the sample rate of the input to that level (hence "dyadic").

The **Lowpass FIR filter coefficients** and **Highpass FIR filter coefficients** parameters specify (respectively) the filter coefficients to be used for every

highpass and lowpass direct-form II transpose filter in the filter bank. The values of these coefficients are typically computed together with the dyadic analysis coefficients using the wavelet family functions in the Wavelet Toolbox (see the Wavelet Toolbox documentation for more information).

Tree Structure

The **Tree structure** parameter specifies an asymmetric (or wavelet) tree, as shown above, or a symmetric structure, as shown below. Note that the *symmetric* structure reconstructs a signal that was *symmetrically* decomposed by the Dyadic Analysis Filter Bank block (i.e., both the high- and low-frequency subbands were divided at each level). The *asymmetric* structure reconstructs a signal that was *asymmetrically* decomposed by the Dyadic Analysis Filter Bank block (i.e., only the low-frequency subbands were divided at each level).

Asymmetric Tree. The asymmetric structure in the first figure (**Tree structure** set to **Asymmetric**) has n+1 inputs, where n is the **Number of levels** parameter value. The sample rate and bandwidth of the output are twice the sample rate and bandwidth of the top input. The sample rate and bandwidth of each additional input (except the last) should be half that of the input to the previous level.

$$T_{si,\,k+1} = 2T_{si,\,k} \qquad 1 \le k < n$$

$$BW_{k+1} = \frac{BW_k}{2} \qquad 1 \le k < n$$

The bottom two inputs (u_n and u_{n+1}) should have the same sample rate and bandwidth since they are processed by the same level.

$$T_{si,\,n+1} = T_{si,\,n}$$

$$BW_{n+1} = BW_n$$

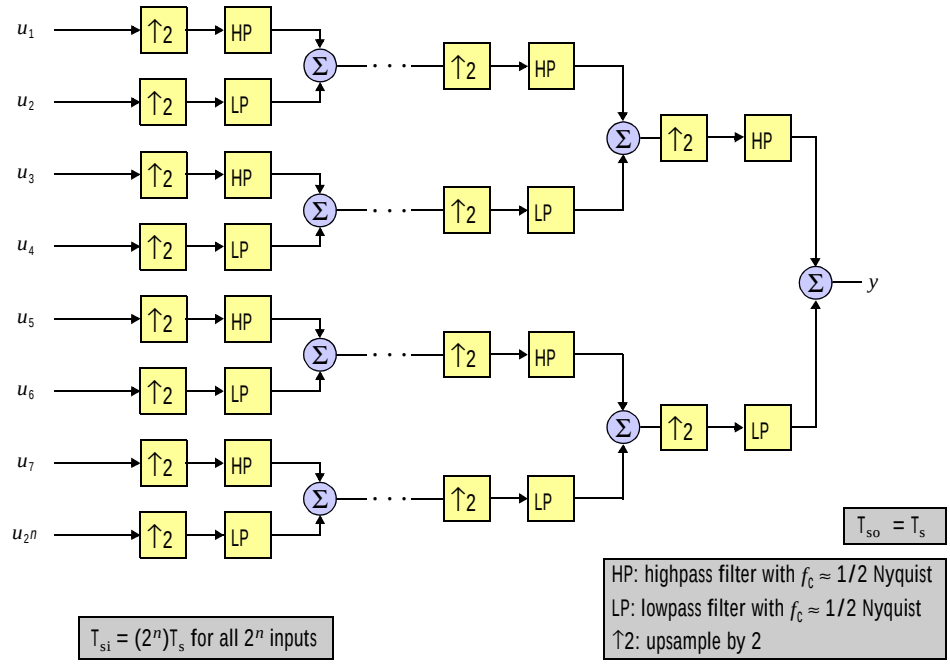
Note that in frame-based mode, the change in the sample period of input u_k is reflected by its frame size, $M_{i,k}$, rather than by its frame rate.

$$M_{i,\,k+1} = \frac{M_{i,\,k}}{2} \qquad 1 \le k < n$$

$$M_{i,\,n+1} = M_{i,\,n}$$

Symmetric Tree. The symmetric structure shown below (**Tree structure** set to **Symmetric**) has 2^n inputs, where n is the **Number of levels** parameter value.

Symmetric Filter Bank, n Levels

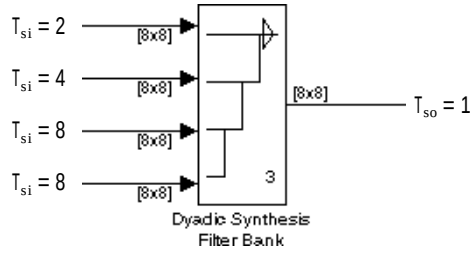


The sample rate and bandwidth of the output are a factor of 2^n higher than the sample rate and bandwidth of the inputs, which are all equal.

Sample-Based Operation

An M-by-N sample-based matrix input is treated as M∗N independent channels, and the block filters each channel independently over time. The output is the same size as the input at each port, one output channel for each input channel. As described earlier, for the asymmetric tree structure, each input port has a different sample period.

The figure below shows the input and output sample periods for the four 64-channel sample-based inputs to a three-level filter bank. The fastest input has a period of 2, so the output period is 1.

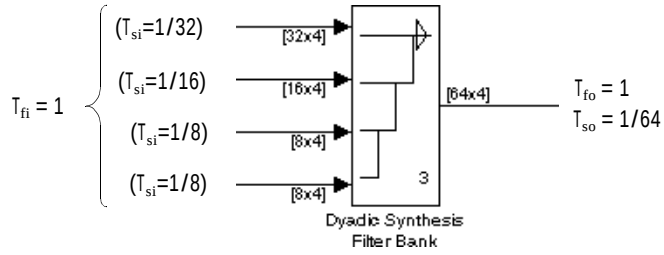


Frame-Based Operation

An M_i-by-N frame-based matrix input is treated as N independent channels, and the block filters each channel independently over time. The number of columns in the output is the same as the number of columns in the input.

All inputs must have the same frame period, which is also the output frame period. The different input sample rates should be represented by the input frame sizes: If the input to the top port has frame size M_i, the input to the second-from-top port should have frame size $M_i/2$, the input to the third-from-top port should have frame size $M_i/4$, and so on. The input to the *bottom* port should have the same frame size as the second-from-bottom port. The increase in the sample rate of the output is also represented by its frame size, which is twice the largest input frame size.

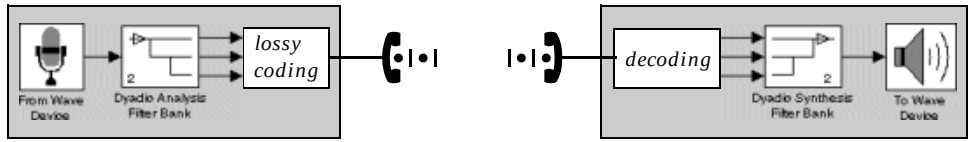
The relationship between sample periods, frame periods, and frame sizes is shown below for a four-channel frame-based input to a three-level filter bank.



Applications

The primary application for asymmetric dyadic synthesis filter banks is coding for compression using wavelets.

At the transmitting end, the output of a 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 the 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.

Latency

Zero Latency. The Dyadic Synthesis Filter Bank block has *no tasking latency* for frame-based operation, which is always single-rate. The block therefore uses the first input samples (received at t=0) to synthesize the first output sample.

Nonzero Latency. The Dyadic Synthesis Filter Bank block has tasking latency only for sample-based operation, which is always multirate. As shown in the table below, the amount of latency, D, depends on the structure (symmetric or asymmetric) of the n-level tree used by the block.

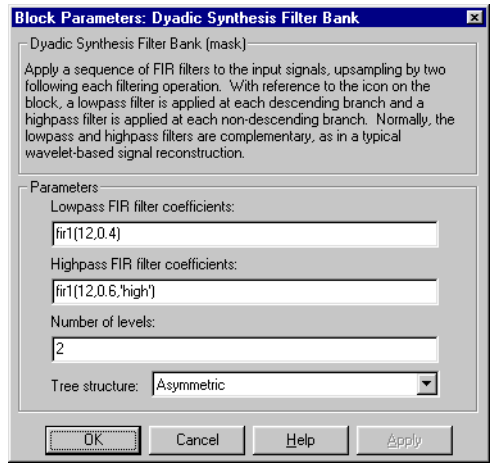
Multirate...	Symmetric Tree	Asymmetric Tree
Single-tasking	None	2^n-2 samples
Multitasking	2^n samples	2^n samples

In the above cases, the block repeats a zero initial condition in each channel for the first D output samples. For example, in single-tasking mode the

asymmetric tree structure generates 2^n-2 zero-valued output samples in each channel before propagating the first synthesized output sample (computed from the inputs received at t=0).

See "Excess Algorithmic Delay (Tasking Latency)" in Chapter 3 and "The Simulation Parameters Dialog Box" in the Simulink documentation for more information about block rates and Simulink's tasking modes.

Dialog Box



Lowpass FIR filter coefficients

A vector of filter coefficients (descending powers of z) to be shared by all the lowpass filters in the filter bank.

Highpass FIR filter coefficients

A vector of filter coefficients (descending powers of z) to be shared by all the highpass filters in the filter bank.

Number of levels

The number of filter bank levels. An n-level asymmetric structure has n+1 inputs; an n-level symmetric structure has 2^n inputs.

Tree structure

The structure of the filter bank, **Asymmetric** (wavelet) or **Symmetric**.

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.

See Also

Dyadic Analysis Filter Bank	DSP Blockset
Wavelet Synthesis	DSP Blockset

See the following sections for related information:

- "Converting Sample Rates and Frame Rates" on page 3-20
- "Multirate Filters" on page 4-24

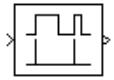
Purpose

Detect a transition of the input from zero to a nonzero value.

Library

Signal Management / Switches and Counters

Description

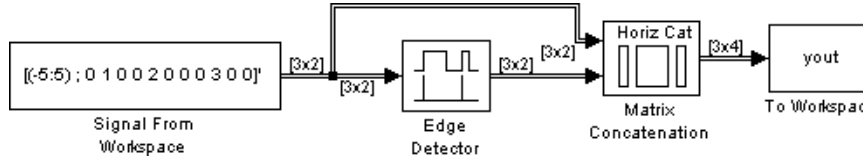


The Edge Detector block generates an impulse (the value `1`) in a given output channel when the corresponding channel of the input transitions from zero to a nonzero value. Otherwise, the block generates zeros in each channel.

The output has the same dimension and sample rate as the input. If the input is frame-based, the output is frame-based; otherwise, the output is sample-based. For frame-based input, an edge that is split across two consecutive frames (i.e., a zero at the bottom of the first frame, and a nonzero value at the top of the following frame) is counted in the frame that contains the nonzero value.

Example

In the model below, the Edge Detector block locates the edges (zero to nonzero transitions) in a two-channel frame-based input with frame size 3. The two input channels are horizontally concatenated with the two output channels to create the four-channel workspace variable `yout`.

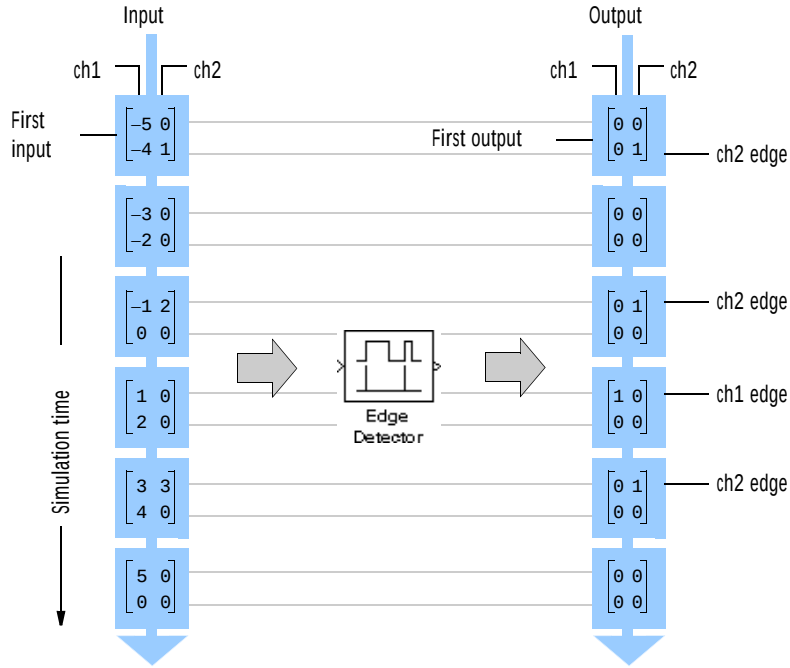


Adjust the block parameters as described below. (Use the default settings for the To Workspace block.)

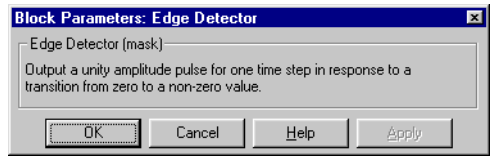
- Set the Signal From Workspace block parameters as follows:
  - **Signal** = `[(-5:5) ; 0 1 0 0 2 0 0 0 3 0 0]'`
  - **Sample time** = `1`
  - **Samples per frame** = `3`
- Set the Matrix Concatenation block parameters as follows:
  - **Number of inputs** = `2`
  - **Concatenation method** = **Horizontal**

As shown below, the block finds edges at sample 7 in channel 1, and at samples 2, 5, and 9 in channel 2.

Edge Detector



Dialog Box



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

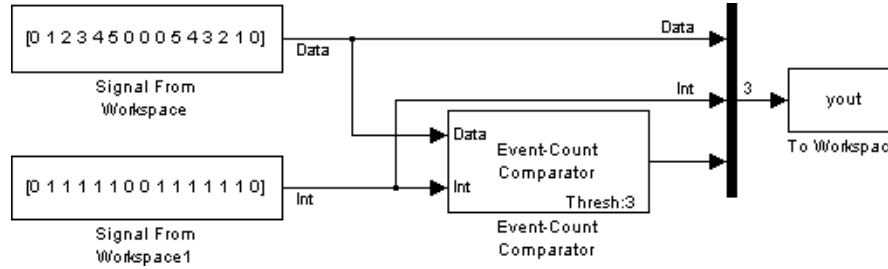
The Event-Count Comparator block records the number of nonzero inputs to the `Data` port during the period that the block is enabled by a high signal (the value `1`) at the interval (`Int`) port. Both inputs must be scalars, and the `Int` input must be sample-based.

When the number of accumulated nonzero inputs first equals the **Event threshold** setting, the block waits one additional sample interval, and then sets the output high (`1`). The block holds the output high until recording is restarted by a low-to-high (`0`-to-`1`) transition at the `Int` port.

If the input to the `Data` port is frame-based, the output is frame-based; otherwise, the output is sample-based.

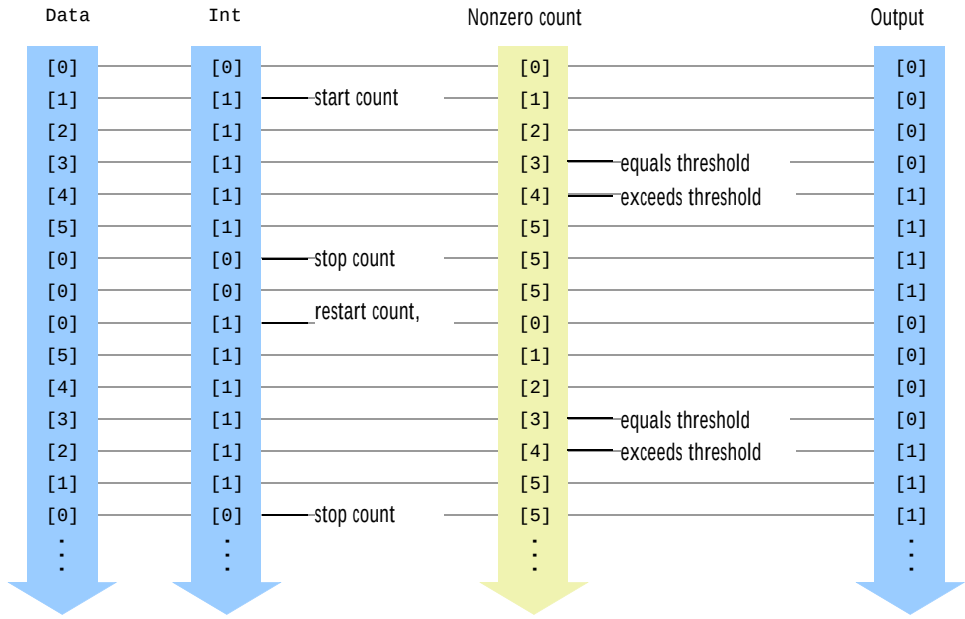
Example

In the model below, the Event-Count Comparator block (**Event threshold** = 3) detects two threshold crossings in the input to the `Data` port, one at sample 4 and one at sample 12.

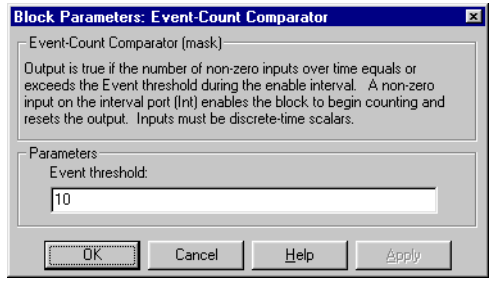


All inputs and outputs are multiplexed into the workspace variable `yout`, whose contents are shown in the figure below. The two left columns in the illustration show the inputs to the `Data` and `Int` ports, the center column shows the state of the block's internal counter, and the right column shows the block's output.

Event-Count Comparator



Dialog Box



Event threshold

The value against which to compare the number of nonzero inputs. Tunable.

See Also

Counter	DSP Blockset
Edge Detector	DSP Blockset

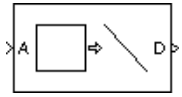
Purpose

Extract the main diagonal of the input matrix.

Library

Math Functions / Matrices and Linear Algebra / Matrix Operations

Description

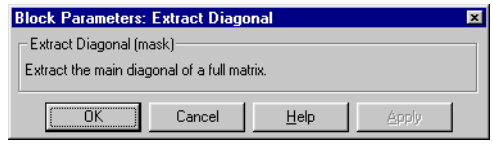


The Extract Diagonal block populates the 1-D output vector with the elements on the main diagonal of the M-by-N input matrix `A`.

```
D = diag(A)        Equivalent MATLAB code
```

The output vector has length `min(M,N)`, and is always sample-based.

Dialog Box



See Also

Constant Diagonal Matrix	DSP Blockset
Create Diagonal Matrix	DSP Blockset
Extract Triangular Matrix	DSP Blockset
`diag`	MATLAB

Extract Triangular Matrix

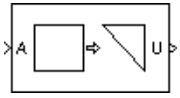
Purpose

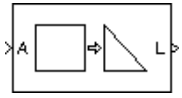
Extract the lower or upper triangle from an input matrix.

Library

Math Functions / Matrices and Linear Algebra / Matrix Operations

Description





The Extract Triangular Matrix block creates a triangular matrix output from the upper or lower triangular elements of an M-by-N input matrix. A length-M 1-D vector input is treated as an M-by-1 matrix.

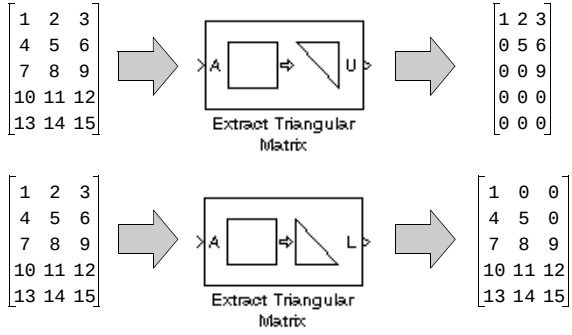
The **Extract** parameter selects between the two components of the input:

- **Upper** – Copies the elements on and above the main diagonal of the input matrix to an output matrix of the same size. The first *row* of the output matrix is therefore identical to the first *row* of the input matrix. The elements below the main diagonal of the output matrix are zero.
- **Lower** – Copies the elements on and below the main diagonal of the input matrix to an output matrix of the same size. The first *column* of the output matrix is therefore identical to the first *column* of the input matrix. The elements above the main diagonal of the output matrix are zero.

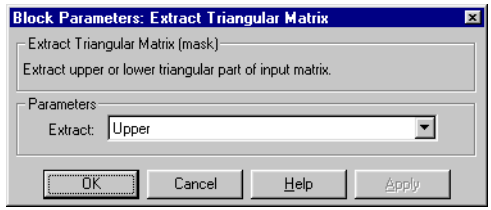
The output has the same frame status as the input.

Example

The example below shows the extraction of upper and lower triangles from a 5-by-3 input matrix.



Dialog Box



Extract

The component of the matrix to copy to the output, upper triangle or lower triangle. Tunable, except in Simulink's external mode.

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

FFT

Purpose

Compute the FFT of the input.

Library

Transforms

Description

The FFT block computes the fast Fourier transform (FFT) of each channel in the M-by-N input matrix, u.

```
y = fft(u,M)        % 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.

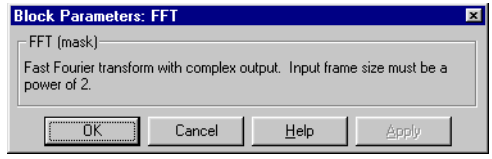
The output is a complex M-by-N matrix whose *l*th column contains the discrete Fourier transform (DFT) of the corresponding input column at M evenly spaced frequency points in the range $[0,F_s)$, where F_s is the input sample rate, $F_s = 1/T_s$ Hz.

$$y(k+1,l) = \sum_{m=0}^{M-1} u(m+1,l)e^{-j2\pi(mk/M)} \qquad k = 0,\ldots,M-1$$

The output is always sample-based, and the output port rate is the same as the input port rate.

For convenience, length-M 1-D vector inputs and *sample-based* length-M row vector inputs are processed as single channels (i.e., as M-by-1 column vectors), and the output has the same dimension as the input.

Dialog Box



See Also

Complex Cepstrum	DSP Blockset
DCT	DSP Blockset
IFFT	DSP Blockset
Zero Pad	DSP Blockset

Filter Realization Wizard

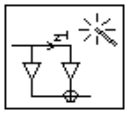
Purpose

Automatically construct filter realizations using Sum, Gain, and Unit Delay blocks.

Library

Filtering / Filter Structures

Description



The Filter Realization Wizard is a tool for automatically creating filter realizations with specific architectures. The Wizard's interface allows you to specify the filter's structure and coefficients, the type of data to be filtered, and optimization criteria for the design. The Wizard then builds the specified filter as a subsystem composed of Sum, Gain, and Unit Delay blocks. You can specify the name of the subsystem ("Filter" is the default) and whether it is placed in the current model or in a new model.

The **Architecture** panel in the Wizard's interface allows you to select from the following realizations.

Architecture	Parameters
Direct-Form I	Numerator, denominator
Direct-Form II	Numerator, denominator
Lattice (AR)	Lattice coefficients
Lattice (MA)	Lattice coefficients
Lattice (ARMA)	Lattice coefficients, ladder coefficients
Symmetric FIR	Coefficients

The **Optimization** panel in the Wizard's interface lets you choose to optimize for zero and unity gains. Zero-gain optimization removes zero-gain paths from the filter structure, and unity-gain optimization substitutes a wire (short circuit) for unity gains.

Type a name for the new filter block in the **Block Name** text field, and select where the block should be placed from the **Destination** pop-up menu. Within a model, the Filter subsystem operates on a sample-based signal (similar to Simulink's Discrete Filter block), filtering each channel over time. Double-click on the subsystem to open it; you can then modify the gains or the filter structure to suit your needs.

Fixed-Point Options

By default, the filter constructed by the Filter Realization Wizard operates using the Simulink standard double-precision arithmetic. If you have the Fixed-Point Blockset installed on your system, you have the additional option of building the filter to operate using single-precision or fixed-point arithmetic. Select the option you want from the **Data Type** panel:

- **Built-in data types**

  The filter is constructed using the standard Simulink Sum, Gain, and Unit Delay blocks, and operates in any precision supported by Simulink (e.g., double-precision, single-precision, Boolean, etc.). This is the default.

- **Single**

  The filter is constructed using the FixPt Sum, FixPt Gain, and FixPt Unit Delay blocks from the Fixed-Point Blockset. The blocks are configured for single-precision arithmetic.

- **Fixed-Point**

  The filter is constructed using the FixPt Sum, FixPt Gain, and FixPt Unit Delay blocks from the Fixed-Point Blockset. The FixPt Sum and FixPt Gain blocks are configured for fixed-point arithmetic using the options specified in the **Fixed-Point** panel of the Filter Realization Wizard. These options include:

  - **Format** (**Signed** or **Unsigned**)
  - **Word size**
  - **Radix pos**
  - **Overflow** (**Wrap** or **Saturate**)
  - **Rounding** (**Zero**, **Nearest**, **Ceiling**, or **Floor**)

For information on these parameters, see the Fixed-Point Blockset documentation.

Examples

The examples below illustrate some of the common architectures available through the Filter Realization Wizard:

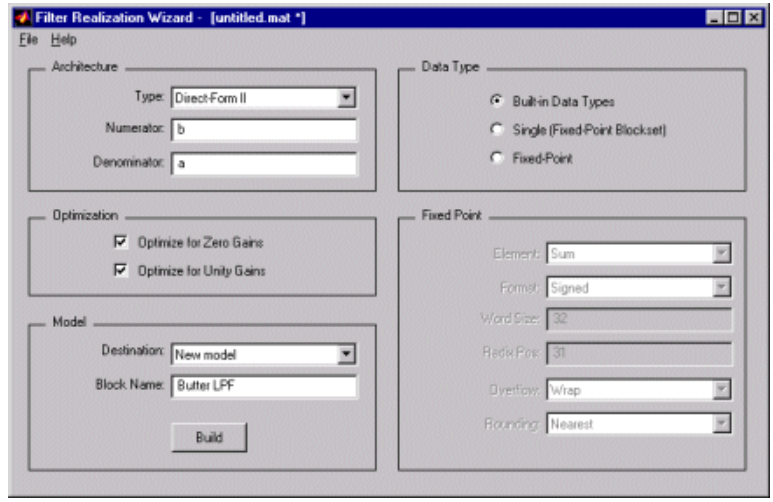
- Example 1: Direct Form II
- Example 2: Second Order Sections
- Example 3: Nth Order Sections
- Example 4: ARMA Lattice

Example 1: Direct Form II

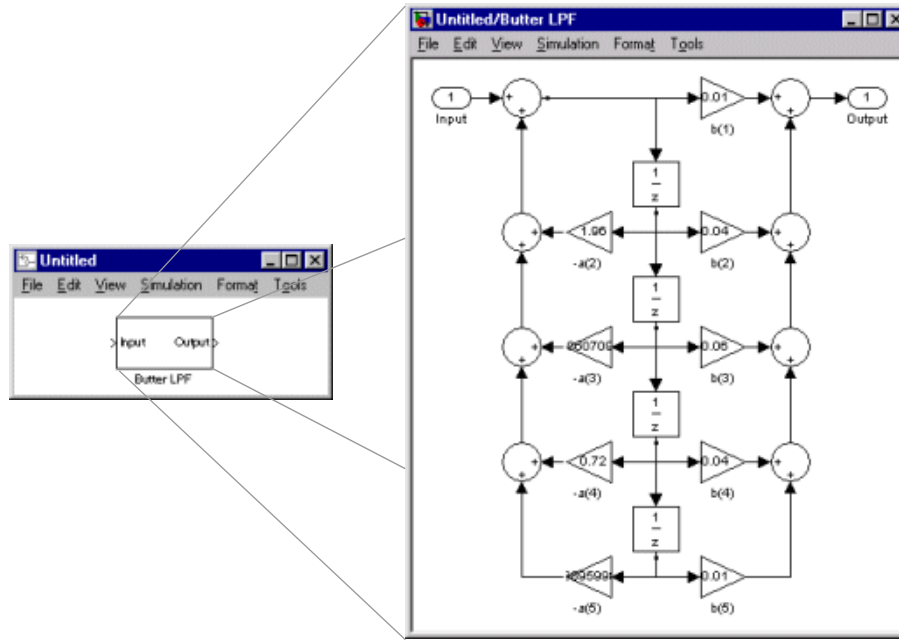
Design an fourth-order, quarter-band, lowpass Butterworth filter:

1. At the MATLAB command line, compute the filter coefficients by entering
   ```
   [b,a] = butter(4,.25);
   ```
2. Launch the Filter Realization Wizard by double-clicking on the icon in the Filter Realizations library.
3. Configure the Wizard to use `b` and `a` as the numerator and denominator of a Direct-Form II structure:
   - Select **Direct-Form II** from the **Type** menu.
   - Type `b` in the **Numerator** text field.
   - Type `a` in the **Denominator** text field.
4. Type a name for the new filter subsystem in the **Block Name** field. The example uses `Butter LPF`.

   The GUI with these settings is shown below.



5. Press the **Build** button to create the specified filter subsystem in a new model window.
6. Double-click the new Butter LPF block to see the Direct-Form II filter realization that the Wizard created.



Example 2: Second Order Sections

Design an eighth-order, quarter-band, lowpass Butterworth filter using second-order sections (SOS):

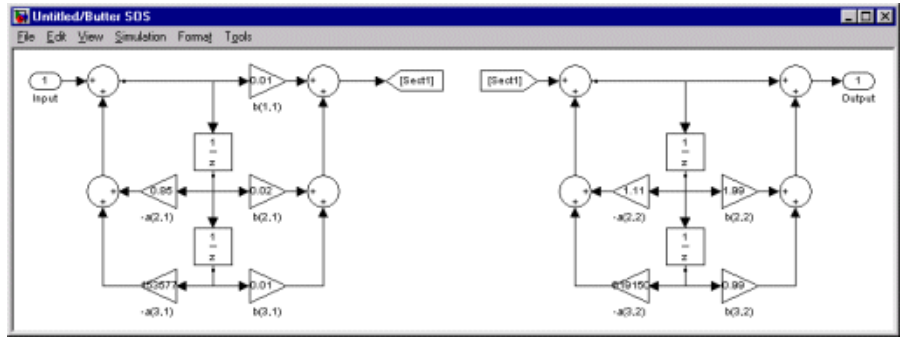
1. At the MATLAB command line, compute the second-order sections by entering

   ```
   [a,b,c,d] = butter(4,.25);
   sos = ss2sos(a,b,c,d);
   ```

2. Configure the Wizard to use `sos` as the numerator of a Direct-Form II structure:
   - Select **Direct-Form II** from the **Type** menu.
   - Type `sos` in the **Numerator** text field.
   - Leave the **Denominator** text field blank.
3. Type a name for the new filter subsystem in the **Block Name** field. The example uses `Butter SOS`.

4 Press the **Build** button to create the specified filter subsystem in a new model window.

5 Double-click the new Butter SOS block to see the Direct-Form II filter realization that the Wizard created.



Note that in a subsystem with the Direct-Form I or II architecture, the filter sections are connected using From and Goto blocks rather than being directly wired together. This makes it easier to recognize and move filter sections in the model window independently of each other.

Example 3: Nth Order Sections

Design a lowpass Butterworth filter using Nth order cascades:

1 At the MATLAB command line, compute the coefficients for the Nth order sections by entering

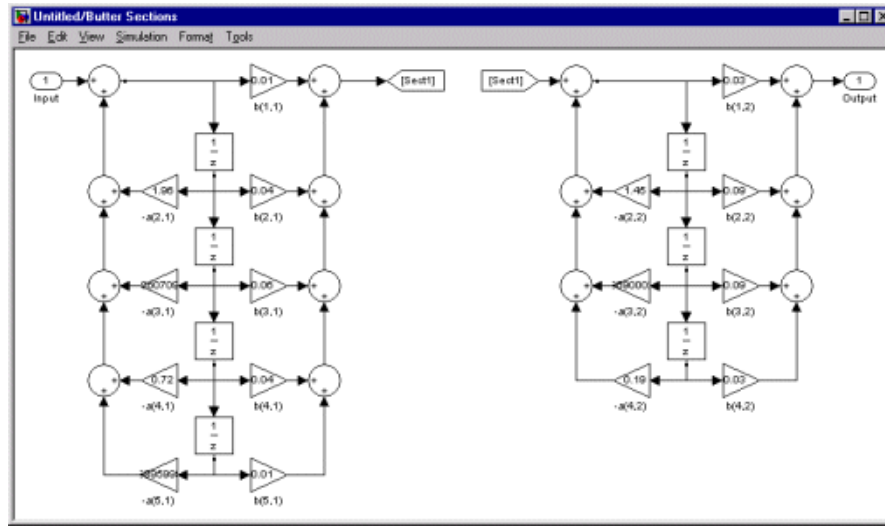
```
[b1,a1] = butter(4,.25);
[b2,a2] = butter(3,.25);
```

2 Configure the Wizard to use these coefficient vectors as the numerator and denominator of a Direct-Form II structure:

- Select **Direct-Form II** from the **Type** menu.
- Type `{b1,b2}` in the **Numerator** text field. Note that the numerator coefficient vector for each section is entered as an element in a cell array. Since this is a two-section filter, a two-cell array is specified in the **Numerator** field. The two filter sections do not need to have the same order.

- Type `{a1,a2}` in the **Denominator** text field. Note that the denominator coefficient vector for each section is also entered as an element in a cell array. Since this is a two-section filter, a two-cell array is specified in the **Denominator** field.

3. Type a name for the new filter subsystem in the **Block Name** field. The example uses `Butter Sections`.
4. Press the **Build** button to create the specified filter subsystem in a new model window.
5. Double-click the new Butter Sections block to see the Direct-Form II filter realization that the Wizard created.



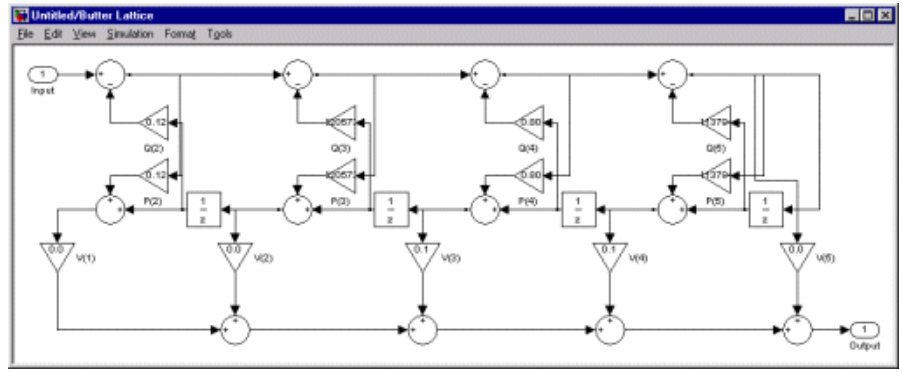
Example 4: ARMA Lattice

Design a fourth-order, quarter-band, lowpass Butterworth filter using an ARMA lattice:

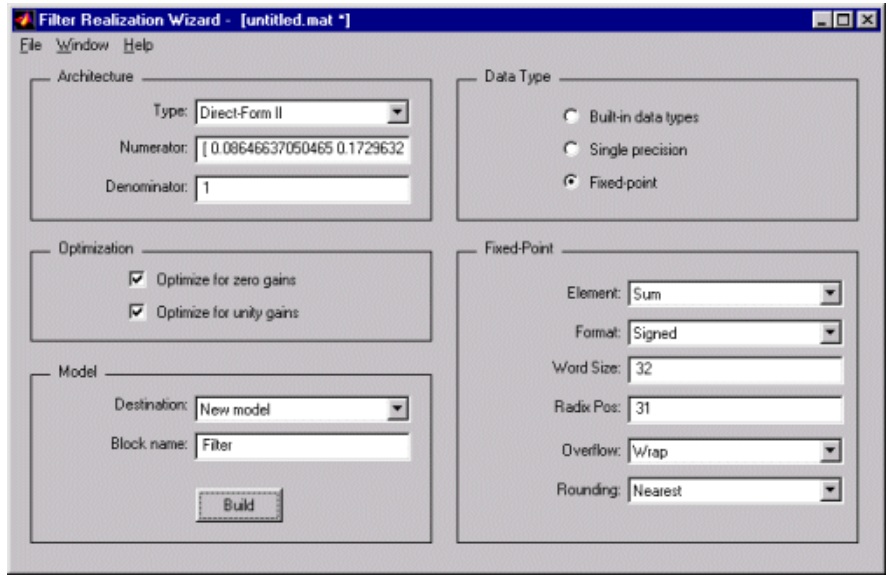
1. At the MATLAB command line, compute the lattice and ladder coefficients (`k` and `v`, respectively) for the ARMA filter.

```
[b,a] = butter(4,.25);
[k,v] = tf2latc(b,a);
```

2 Configure the Wizard to use k and v as the coefficients of the lattice design:
   - Select **Lattice (ARMA)** from the **Type** menu.
   - Type k in the **Lattice Coeffs** text field.
   - Type v in the **Ladder Coeffs** text field.
3 Type a name for the new filter subsystem in the **Block Name** field. The example uses `Butter Lattice`.
4 Press the **Build** button to create the specified filter subsystem in a new model window.
5 Double-click the new Butter Lattice block to see the ARMA filter realization that the Wizard created.



Dialog Box



The parameters displayed in the **Architecture** panel vary for different selections in the **Type** menu. Only a portion of the parameters listed below are visible in the wizard at any one time.

Type

The filter architecture: **Direct-Form I**, **Direct-Form II**, **Symmetric FIR**, **Lattice (MA)**, **Lattice (AR)**, **Lattice (ARMA)**.

Numerator

The numerator coefficients for the direct-form I and II structures, specified as a vector or variable name.

Denominator

The denominator coefficients for the direct-form I and II structures, specified as a vector or variable name.

Coefficients

The coefficients for the symmetric FIR structure, specified as a vector or variable name.

Lattice Coeffs

The lattice coefficients for the lattice MA/AR/ARMA structures, specified as a vector or variable name.

Ladder Coeffs

The ladder coefficients for the lattice ARMA structure, specified as a vector or variable name.

Optimize for zero gains

Enables zero-gain optimization (when checked) by removing zero-gain paths from the filter structure.

Optimize for unity gains

Enables unity-gain optimization (when checked) by substituting a wire (short circuit) for unity gains.

Destination

The location where the new filter block should be created.

Block name

The name of the new filter block.

Build

Generate the filter.

Data type

The precision of the data that the filter will process. **Built-in data types**, when selected, configures the block to build the filter using double-precision Simulink blocks. **Single precision** and **Fixed-point** configure the block to build the filter using Fixed-Point Blockset blocks.

Fixed-point

Options for fixed-point filter construction. See the Fixed-Point Blockset documentation.

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.

See Also

Biquadratic Filter	DSP Blockset
Direct-Form II Transpose Filter	DSP Blockset
Discrete Filter	Simulink
Time-Varying Direct-Form II Transpose Filter	DSP Blockset
Time-Varying Lattice Filter	DSP Blockset
`filter`	MATLAB

See “Filter Structures” on page 4-23 for related information.

FIR Decimation

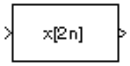
Purpose

Filter and downsample an input signal.

Library

Filtering / Multirate Filters

Description



The FIR Decimation block resamples the discrete-time input at a rate K times slower than the input sample rate, where the integer K is specified by the **Decimation factor** parameter. This process consists of two steps:

- The block filters the input data using a direct-form II transpose FIR filter.
- The block downsamples the filtered data to a lower rate by discarding K-1 consecutive samples following every sample retained.

The FIR Decimation block implements the above FIR filtering and downsampling steps together using a polyphase filter structure, which is more efficient than straightforward filter-then-decimate algorithms. The output of the decimator is the first phase of the polyphase filter.

The **FIR filter coefficients** parameter specifies the numerator coefficients of the FIR filter transfer function H(*z*).

$$H(z) = B(z) = b_1 + b_2 z^{-1} + \ldots + b_m z^{-(m-1)}$$

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.

Sample-Based Operation

An M-by-N sample-based matrix input is treated as M∗N independent channels, and the block decimates each channel over time. The output sample period is K times longer than the input sample period ($T_{so} = KT_{si}$), and the input and output sizes are identical.

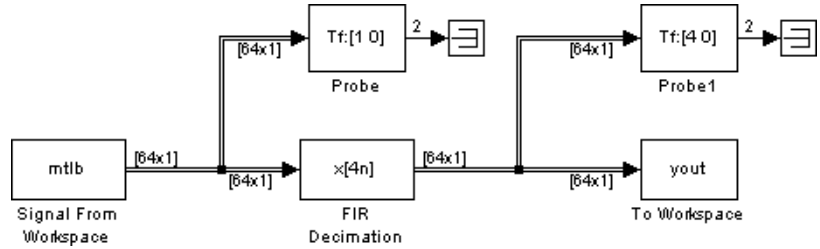
Frame-Based Operation

An M_i-by-N frame-based matrix input is treated as N independent channels, and the block decimates each channel over time. The **Framing** parameter determines how the block adjusts the rate at the output to accommodate the reduced number of samples. There are two available options:

- **Maintain input frame size**

  The block generates the output at the slower (decimated) rate by using a proportionally longer frame *period* at the output port than at the input port. For decimation by a factor of K, the output frame period is K times longer than the input frame period ($T_{fo} = KT_{fi}$), but the input and output frame sizes are equal.

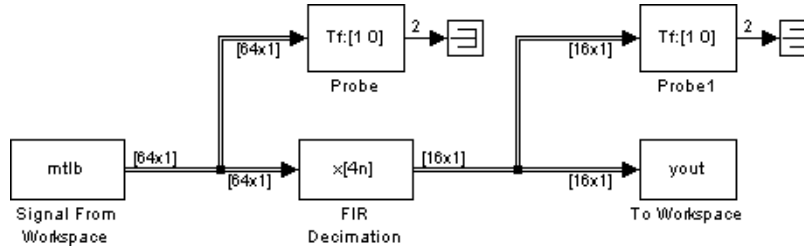
  The example below shows a single-channel input with a frame period of 1 second (**Sample time** = `1/64` and **Samples per frame** = `64` in the Signal From Workspace block) being decimated by a factor of 4 to a frame period of 4 seconds. The input and output frame sizes are identical.



- **Maintain input frame rate**

  The block generates the output at the slower (decimated) rate by using a proportionally smaller frame *size* than the input. For decimation by a factor of K, the output frame size is K times smaller than the input frame size ($M_o = M_i/K$), but the input and output frame rates are equal. *The input frame size,* M_i*, must be a multiple of the decimation factor,* K.

  The example below shows a single-channel input of frame size 64 being decimated by a factor of 4 to a frame size of 16. The block's input and output frame rates are identical.



Latency

Zero Latency. The FIR Decimation block has *zero tasking latency* for all single-rate operations. The block is single-rate for the particular combinations of sampling mode and parameter settings shown in the table below.

Sampling Mode	Parameter Settings
Sample-based	**Decimation factor** parameter, K, is 1.
Frame-based	**Decimation factor** parameter, K, is 1, *or* **Framing** parameter is **Maintain input frame rate**.

Note that in sample-based mode, single-rate operation occurs only in the trivial case of factor-of-1 decimation.

The block also has zero latency for sample-based multirate operations in Simulink's 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 Simulink's tasking mode and the block's sampling mode, as shown in the table below.

Multirate...	Sample-Based Latency	Frame-Based Latency
Single-tasking	None	One frame (M_i samples)
Multitasking	One sample	One frame (M_i samples)

In cases of *one-sample latency*, a zero initial condition appears as the first output sample in each channel. The first filtered input sample appears as the second output sample, followed by filtered input samples K+1, 2K+1, and so on.

In cases of *one-frame latency*, the first M_i output rows contain zeros, where M_i is the input frame size. The first filtered input sample (first filtered row of the input matrix) appears in the output as sample M_i+1, followed by filtered input

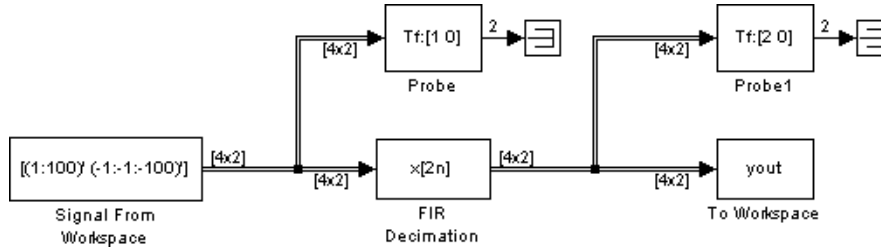
samples K+1, 2K+1, and so on. See the example below for an illustration of this case.

See “Excess Algorithmic Delay (Tasking Latency)” on page 3-91 and “The Simulation Parameters Dialog Box” in the Simulink documentation for more information about block rates and Simulink's tasking modes.

Examples

Example 1

Construct the frame-based model shown below.



Adjust the block parameters as follows:

- Configure the Signal From Workspace block to generate a two-channel signal with frame size of 4 and sample period of 0.25. This represents an output frame period of 1 (0.25∗4). The first channel should contain the positive ramp signal 1, 2, ..., 100, and the second channel should contain the negative ramp signal -1, -2, ..., -100.
  - **Signal** = `[(1:100)' (-1:-1:-100)']`
  - **Sample time** = `0.25`
  - **Samples per frame** = `4`
- Configure the FIR Decimation block to decimate the two-channel input by decreasing the output frame rate by a factor of 2 relative to the input frame rate. Use a third-order filter with normalized cutoff frequency, f_{n0}, of 0.25. (Note that f_{n0} satisfies $f_{n0} \le 1/K$.)
  - **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 deselecting 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 Simulink's multitasking mode, select **Fixed-step** and **discrete** from the **Type** controls in the **Solver** panel of the **Simulation Parameters** dialog box, and select **MultiTasking** from the **Mode** parameter. Also set the **Stop time** to `30`.

Run the model and look at the output, `yout`. The first few samples of each channel are shown below.

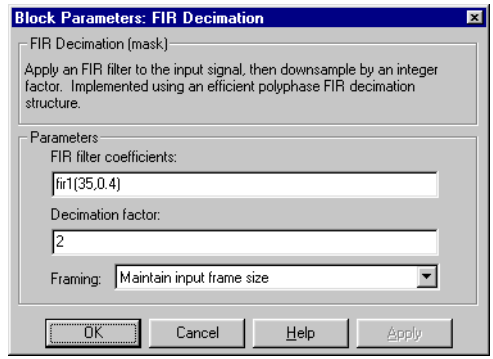
```
yout =

         0         0
         0         0
         0         0
         0         0
    0.0386   -0.0386
    1.5000   -1.5000
    3.5000   -3.5000
    5.5000   -5.5000
    7.5000   -7.5000
    9.5000   -9.5000
   11.5000  -11.5000
```

Since we ran this frame-based multirate model in multitasking mode, the first four (M_i) output rows are zero. The first filtered input matrix row appears in the output as sample 5 (i.e., sample M_i+1).

Example 2

The `dspmrf_menu` demo illustrates the use of the FIR Decimation block in a number of multistage multirate filters.

Dialog Box



FIR filter coefficients

The lowpass FIR filter coefficients, in descending powers of *z*.

Decimation factor

The integer factor, K, by which to decrease the sample rate of the input sequence.

Framing

For frame-based operation, the method by which to implement the decimation; reduce the output frame rate, or reduce the output frame size.

See Also

Direct-Form II Transpose Filter	DSP Blockset
Downsample	DSP Blockset
FIR Interpolation	DSP Blockset
FIR Rate Conversion	DSP Blockset
`decimate`	Signal Processing Toolbox
`fir1`	Signal Processing Toolbox
`fir2`	Signal Processing Toolbox
`firls`	Signal Processing Toolbox
`remez`	Signal Processing Toolbox

See the following sections for related information:

- “Converting Sample Rates and Frame Rates” on page 3-20
- “Multirate Filters” on page 4-24

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 II transpose FIR filter.

The FIR Interpolation block implements the above upsampling and FIR filtering steps together using a polyphase filter structure, which is more efficient than straightforward upsample-then-filter algorithms.

The **FIR filter coefficients** parameter specifies the numerator coefficients of the FIR filter transfer function H(*z*).

$$H(z) = B(z) = b_1 + b_2 z^{-1} + \ldots + b_m z^{-(m-1)}$$

The coefficient vector, `[b(1) b(2) ... b(m)]`, can be generated by one of the filter design functions in the Signal Processing Toolbox (such as `fir1`), and should have a length greater than the interpolation factor (*m*>L). The filter should be lowpass with normalized cutoff frequency no greater than 1/L. All filter states are internally initialized to zero.

Sample-Based Operation

An M-by-N sample-based matrix input is treated as M∗N independent channels, and the block interpolates each channel over time. The output sample period is L times shorter than the input sample period ($T_{so} = T_{si}/L$), and the input and output sizes are identical.

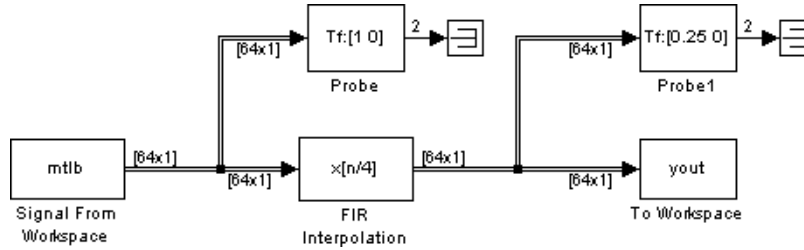
Frame-Based Operation

An M_i-by-N frame-based matrix input is treated as N independent channels, and the block decimates each channel over time. The **Framing** parameter determines how the block adjusts the rate at the output to accommodate the added samples. There are two available options:

- **Maintain input frame size**

  The block generates the output at the faster (interpolated) rate by using a proportionally shorter frame *period* at the output port than at the input port. For interpolation by a factor of L, the output frame period is L times shorter than the input frame period ($T_{fo} = T_{fi}/L$), but the input and output frame sizes are equal.

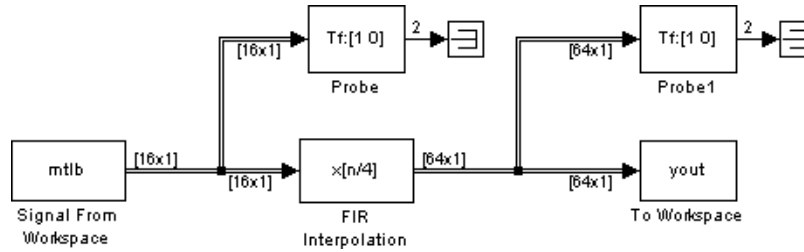
  The example below shows a single-channel input with a frame period of 1 second (**Sample time** = `1/64` and **Samples per frame** = `64` in the Signal From Workspace block) being interpolated by a factor of 4 to a frame period of 0.25 seconds. The input and output frame sizes are identical.



- **Maintain input frame rate**

  The block generates the output at the faster (interpolated) rate by using a proportionally larger frame *size* than the input. For interpolation by a factor of L, the output frame size is L times larger than the input frame size ($M_o = M_i * L$), but the input and output frame rates are equal.

  The example below shows a single-channel input of frame size 16 being interpolated by a factor of 4 to a frame size of 64. The block's input and output frame rates are identical.



Latency

Zero Latency. The FIR Interpolation block has *zero tasking latency* for all single-rate operations. The block is single-rate for the particular combinations of sampling mode and parameter settings shown in the table below.

Sampling Mode	Parameter Settings
Sample-based	**Interpolation factor** parameter, L, is 1.
Frame-based	**Interpolation factor** parameter, L, is 1, *or* **Framing** parameter is **Maintain input frame rate**.

Note that in sample-based mode, single-rate operation occurs only in the trivial case of factor-of-1 interpolation.

The block also has zero latency for sample-based multirate operations in Simulink's 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 Simulink's tasking mode and the block's sampling mode, as shown in the table below.

Multirate...	Sample-Based Latency	Frame-Based Latency
Single-tasking	None	One frame (M_i samples)
Multitasking	One sample	One frame (M_i samples)

In cases of *one-sample latency*, a zero initial condition appears as the first output sample in each channel, followed immediately by the first filtered input sample, L-1 interpolated values, and so on.

In cases of *one-frame latency*, the first M_i output rows contain zeros, where M_i is the input frame size. The first filtered input sample (first filtered row of the

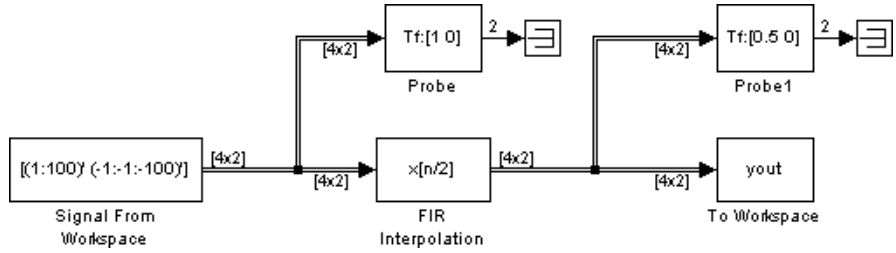
input matrix) appears in the output as sample M_i+1, followed by L-1 interpolated values, the second filtered input sample, and so on. See the example below for an illustration of this case.

See “Excess Algorithmic Delay (Tasking Latency)” on page 3-91 and “The Simulation Parameters Dialog Box” in the Simulink documentation for more information about block rates and Simulink’s tasking modes.

Example

Example 1

Construct the frame-based model shown below.



Adjust the block parameters as follows.

- Configure the Signal From Workspace block to generate a two-channel signal with frame size of 4 and sample period of 0.25. This represents an output frame period of 1 (0.25∗4). The first channel should contain the positive ramp signal 1, 2, ..., 100, and the second channel should contain the negative ramp signal -1, -2, ..., -100.
  - **Signal** = `[(1:100)' (-1:-1:-100)']`
  - **Sample time** = `0.25`
  - **Samples per frame** = `4`
- Configure the FIR Interpolation block to interpolate the two-channel input by increasing the output frame rate by a factor of 2 relative to the input frame rate. Use a third-order filter (m=3) with normalized cutoff frequency, f_{n0}, of 0.25. (Note that f_{n0} and m satisfy $f_{n0} \le 1/L$ 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 deselecting 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 Simulink's multitasking mode, select **Fixed-step** and **discrete** from the **Type** controls in the **Solver** panel of the **Simulation Parameters** dialog box, and select **MultiTasking** from the **Mode** parameter. Also set the **Stop time** to `30`.

Run the model and look at the output, `yout`. The first few samples of each channel are shown below.

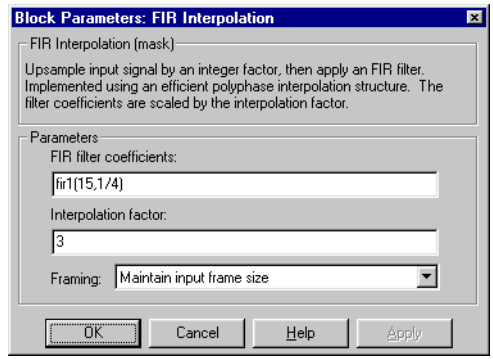
```
yout =

         0         0
         0         0
         0         0
         0         0
    0.0386   -0.0386
    0.4614   -0.4614
    0.5386   -0.5386
    0.9614   -0.9614
    1.0386   -1.0386
```

Since we ran this frame-based multirate model in multitasking mode, the first four (M_i) output rows are zero. The first filtered input matrix row appears in the output as sample 5 (i.e., sample M_i+1). Every second row is an interpolated value.

Example 2

The `dspintrp` demo provides another simple example, and the `dspmrf_menu` demo illustrates the use of the FIR Interpolation block in a number of multistage multirate filters.

Dialog Box



FIR filter coefficients

The FIR filter coefficients, in descending powers of *z*.

Interpolation factor

The integer factor, L, by which to increase the sample rate of the input sequence.

Framing

For frame-based operation, the method by which to implement the interpolation: increase the output frame rate, or increase the output frame size.

See Also

FIR Decimation	DSP Blockset
FIR Rate Conversion	DSP Blockset
Upsample	DSP Blockset
`fir1`	Signal Processing Toolbox
`fir2`	Signal Processing Toolbox
`firls`	Signal Processing Toolbox
`interp`	Signal Processing Toolbox
`remez`	Signal Processing Toolbox

See the following sections for related information:

- "Converting Sample Rates and Frame Rates" on page 3-20
- "Multirate Filters" on page 4-24

FIR Rate Conversion

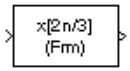
Purpose

Upsample, filter, and downsample an input signal.

Library

Filtering / Multirate Filters

Description



The FIR Rate Conversion block resamples the discrete-time input to a period K/L times the input sample period, where the integer K is specified by the **Decimation factor** parameter and the integer L is specified by the **Interpolation factor** parameter. The resampling process consists of the following steps:

- The block upsamples the input to a higher rate by inserting L-1 zeros between input samples.
- The upsampled data is passed through a direct-form II transpose FIR filter.
- The block downsamples the filtered data to a lower rate by discarding K-1 consecutive samples following each sample retained.

K and L must be *relatively prime* integers; that is, the ratio K/L cannot be reducible to a ratio of smaller integers. The FIR Rate Conversion block implements the above three steps together using a polyphase filter structure, which is more efficient than straightforward upsample-filter-decimate algorithms. The output of the interpolator is the first filter phase, while the output of the decimator is the last filter phase. When both K and L are greater than 1, the resulting output is the last decimation phase from the first interpolation phase.

The **FIR filter coefficients** parameter specifies the numerator coefficients of the FIR filter transfer function H(*z*).

$$H(z) = B(z) = b_1 + b_2 z^{-1} + \ldots + b_m z^{-(m-1)}$$

The coefficient vector, `[b(1) b(2) ... b(m)]`, can be generated by one of the filter design functions in the Signal Processing Toolbox (such as `fir1`), and should have a length greater than the interpolation factor (*m*>L). The filter should be lowpass with normalized cutoff frequency no greater than `min(1/L,1/K)`. All filter states are internally initialized to zero.

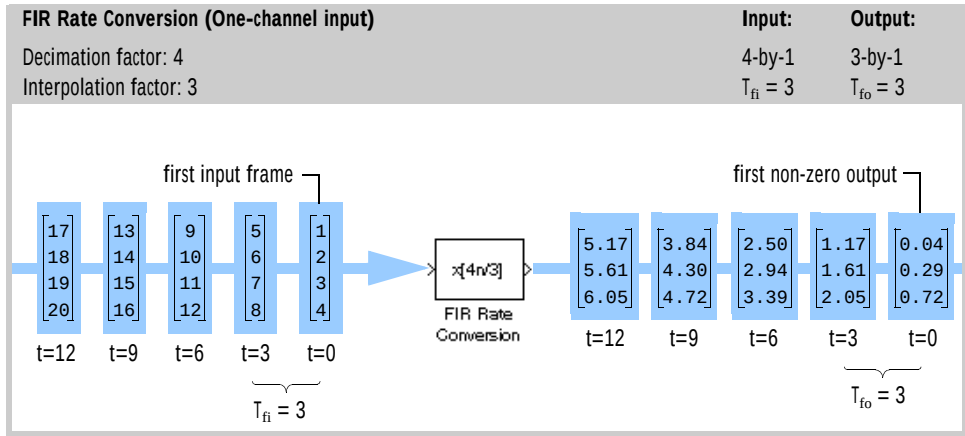
Frame-Based Operation

This block accepts *only* frame-based inputs. An M_i-by-N frame-based matrix input is treated as N independent channels, and the block resamples each channel independently over time.

The **Interpolation factor**, L, and **Decimation factor**, K, must satisfy the relation

$$\frac{K}{L} = \frac{M_i}{M_o}$$

for an *integer* output frame size M_o. The simplest way to satisfy this requirement is to let the **Decimation factor** equal the input frame size, M_i. The output frame size, M_o, is then equal to the **Interpolation factor**. This change in the frame size, from M_i to M_o, produces the desired rate conversion while leaving the output frame period the same as the input ($T_{fo} = T_{fi}$).



Latency

The FIR Rate Conversion block has no tasking latency. The block propagates the first filtered input (received at *t*=0) as the first output sample.

Examples

The dspsrcnv demo compares sample rate conversion performed by the FIR Rate Conversion block with the same conversion performed by a cascade of Upsample, Direct-Form II Transpose Filter, and Downsample blocks.

Diagnostics

An error is generated if the relation between K and L shown above is not satisfied.

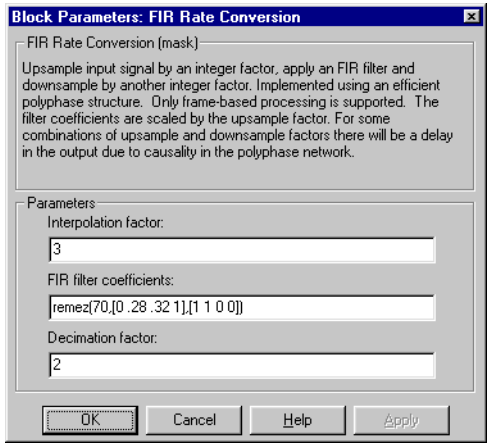
```
(Input port width)/(Output port width) must equal the
(Decimation factor)/(Interpolation factor).
```

A warning is generated if L and K are not relatively prime; that is, if the ratio L/K can be reduced to a ratio of smaller integers.

```
Warning: Integer conversion factors are not relatively prime in
block 'modelname/FIR Rate Conversion (Frame)'. Converting ratio
L/M to l/m.
```

The block scales the ratio to be relatively prime, and continues the simulation.

Dialog Box



Interpolation factor

The integer factor, L, by which to upsample the signal before filtering.

FIR filter coefficients

The FIR filter coefficients, in descending powers of *z*.

Decimation factor

The integer factor, K, by which to downsample the signal after filtering.

References

Fliege, N. J. *Multirate Digital Signal Processing: Multirate Systems, Filter Banks, Wavelets*. West Sussex, England: John Wiley & Sons, 1994.

See Also

Downsample	DSP Blockset
FIR Decimation	DSP Blockset
FIR Interpolation	DSP Blockset
Upsample	DSP Blockset
`fir1`	Signal Processing Toolbox
`fir2`	Signal Processing Toolbox
`firls`	Signal Processing Toolbox
`remez`	Signal Processing Toolbox
`upfirdn`	Signal Processing Toolbox

See the following sections for related information:

- "Converting Sample Rates and Frame Rates" on page 3-20
- "Multirate Filters" on page 4-24

Flip

Purpose

Flip the input vertically or horizontally.

Library

Signal Management / Indexing

Description

The Flip block vertically or horizontally reverses the M-by-N input matrix, u. The output always has the same dimension and frame status as the input.

When **Columns** is selected from the **Flip along** menu, the block *vertically* flips the input so that the first row of the input is the last row of the output.

```
y = flipud(u)          % Equivalent MATLAB code
```

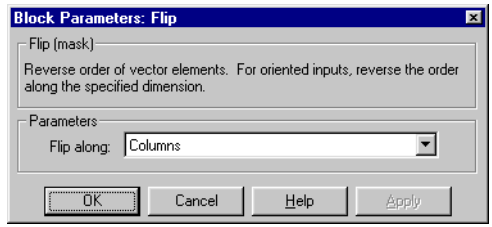
For convenience, length-M 1-D vector inputs are treated as M-by-1 column vectors for vertical flipping.

When **Rows** is selected from the **Flip along** menu, the block *horizontally* flips the input so that the first column of the input is the last column of the output.

```
y = fliplr(u)          % Equivalent MATLAB code
```

For convenience, length-N 1-D vector inputs are treated as 1-by-N row vectors for horizontal flipping. The output always has the same dimension and frame status as the input.

Dialog Box



Flip along

The dimension along which to flip the input. **Columns** specifies vertical flipping, while **Rows** specifies horizontal flipping.

See Also

Selector	Simulink
Transpose	DSP Blockset
Variable Selector	DSP Blockset
`flipud`	MATLAB
`fliplr`	MATLAB

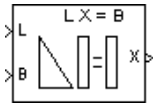
Purpose

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

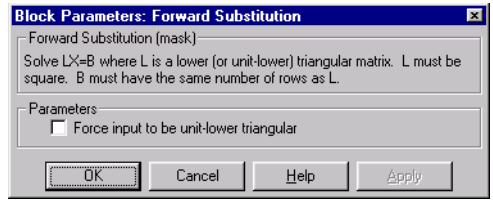


The Forward Substitution block solves the linear system L*X*=B by simple forward substitution of variables, where L is the lower triangular M-by-M matrix input to the L port, and B is the M-by-N matrix input to the B port. The output is the solution of the equations, the M-by-N matrix *X*, and is always sample-based.

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

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

Dialog Box



Force input to be unit-lower triangular

Replaces the elements on the diagonal of L with 1s when selected. Tunable.

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 4-31 for related information.

Frame Status Conversion

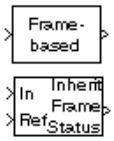
Purpose

Specify the frame status of the output, sample-based or frame-based.

Library

Signal Management / Signal Attributes

Description

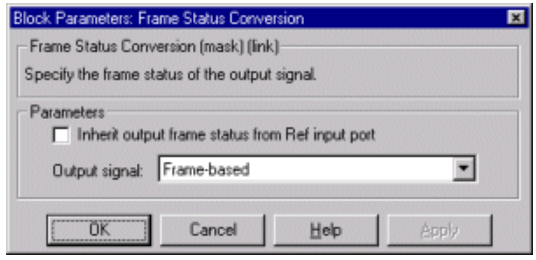


The Frame Status Conversion block passes the input through to the output, and sets the output frame status to the **Output signal** parameter, which can be either **Frame-based** or **Sample-based**. The output frame status can also be inherited from the signal at the `Ref` (reference) input port, which is made visible by selecting the **Inherit output frame status from Ref input port** check box.

If the **Output signal** parameter setting or the inherited signal's frame status differs from the input frame status, the block changes the input frame status accordingly, but does not otherwise alter the signal. In particular, the block does not rebuffer or resize 2-D inputs. Because 1-D vectors cannot be frame-based, if the input is a length-M 1-D vector, and the **Output signal** parameter is set to **Frame-based**, the output is a frame-based M-by-1 matrix (i.e., a single channel).

If the **Output signal** parameter or the inherited signal's frame status matches the input frame status, the block passes the input through to the output unaltered.

Dialog Box



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

Purpose

Read audio data from a standard audio device in real-time. (Windows 95/98/NT only)

Library

DSP Sources

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, deselect **Use default audio device**, and enter the desired device identification number in the **Audio device ID** parameter. The device ID is an integer value that the block associates with the sound device. A three-device system, for example, has device ID numbers of 1, 2, and 3.

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 deselected. 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 amplitude of the input from the sound device should be in the range ±1. Values outside this range are clipped to the nearest allowable value. If the audio signal is saturating at ±1, you can reduce the microphone gain from the **Multimedia Properties** window (available through the Windows 95/98/NT **Control Panel**). The audio data is processed in uncompressed PCM (pulse code modulation) format, and should typically be sampled at one of the standard Windows audio device rates: 8000, 11025, 22050, or 44100 Hz. You can select one of these rates from the **Sample rate** parameter. To specify a different rate, select the **User-defined** option and enter a value in the **User-defined sample rate** parameter.

The **Sample Width (bits)** parameter specifies the number of bits used to represent the signal samples read by the audio device. Two settings are available:

- **8** – allocates 8 bits to each sample, allowing a resolution of 256 levels
- **16** – allocates 16 bits to each sample, allowing a resolution of 65536 levels

The 16-bit sample width setting requires more memory but yields better fidelity. The output from the block is independent of the **Sample Width (bits)** setting, and is always double precision.

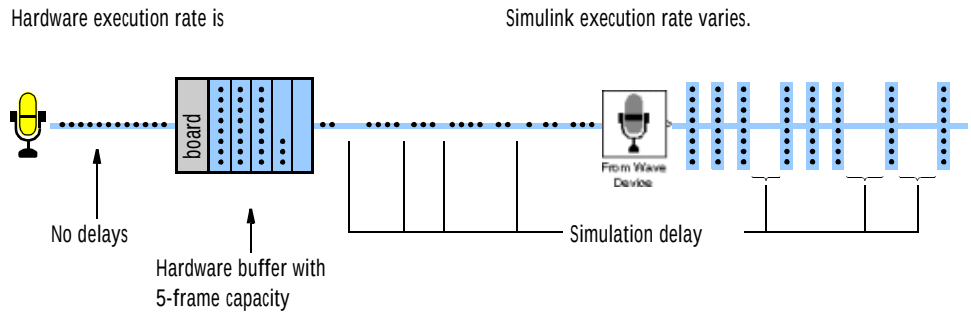
Buffering

Since the audio device accepts real-time audio input, Simulink must read a continuous stream of data from the device throughout the simulation. Delays in reading data from the audio hardware can result in hardware errors or distortion of the signal. This means that the From Wave Device block must in principle read data from the audio hardware as quickly as the hardware itself acquires the signal. However, the block often *cannot* match the throughput rate of the audio hardware, especially when the simulation is running from within Simulink rather than as generated code. (Simulink operations are generally slower than comparable hardware operations, and execution speed routinely varies during the simulation as the host operating system services other processes.) The block must therefore rely on a buffering strategy to ensure that signal data can be read on schedule without losing samples.

At the start of the simulation, the audio device begins writing the input data to a (hardware) buffer with a capacity of T_b seconds. The From Wave Device block immediately begins pulling the earliest samples off the buffer (first in, first out) and collecting them in length-M frames for output. As the audio device continues to append inputs to the bottom of the buffer, the From Wave Device block continues to pull inputs off the top of the buffer at the best possible rate.

The following figure shows an audio signal being acquired and output with a frame size of 8 samples. The buffer of the sound board is approaching its five-frame capacity at the instant shown, which means that the hardware is adding samples to the buffer more rapidly than the block is pulling them off. (If

the signal sample rate was 8 kHz, this small buffer could hold approximately 0.005 second of data.)



If the simulation throughput rate is higher than the hardware throughput rate, the buffer remains empty throughout the simulation. If necessary, the From Wave Device block simply waits for new samples to become available on the buffer (the block does not interpolate between samples). More typically, the simulation throughput rate is lower than the hardware throughput rate, and the buffer tends to fill over the duration of the simulation.

Troubleshooting

If the buffer size is too small in relation to the simulation throughput rate, the buffer may fill before the entire length of signal is processed. This usually results in a device error or undesired device output. When this problem occurs, you can choose to either increase the buffer size or the simulation throughput rate:

- *Increase the buffer size*

  The **Queue duration** parameter specifies the duration of signal, T_b (in real-time seconds), that can be buffered in hardware during the simulation. Equivalently, this is the maximum length of time that the block's data acquisition can lag the hardware's data acquisition. The number of frames buffered is approximately

  $$\frac{T_b F_s}{M}$$

  where F_s is the sample rate of the signal and M is the number of samples per frame. The required buffer size for a given signal depends on the signal

length, the frame size, and the speed of the simulation. Note that increasing the buffer size may increase model latency.

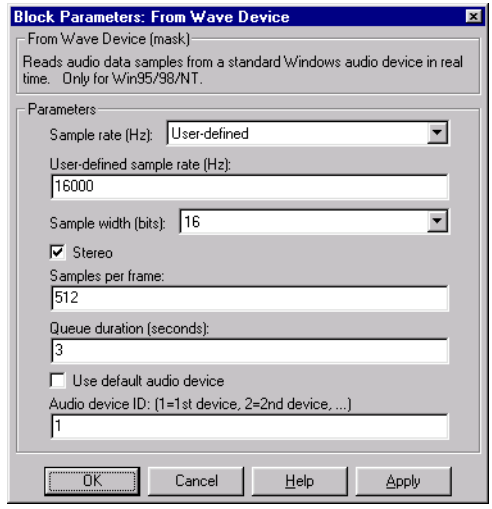
- *Increase the simulation throughput rate*

  Two useful methods for improving simulation throughput rates are increasing the signal frame size and compiling the simulation into native code:

  - Increase frame sizes (and convert sample-based signals to frame-based signals) throughout the model to reduce the amount of block-to-block communication overhead. This can drastically increase throughput rates in many cases. However, larger frame sizes generally result in greater model latency due to initial buffering operations.
  - Generate executable code with Real Time Workshop. Native code runs much faster than Simulink, and should provide rates adequate for real-time audio processing.

More general ways to improve throughput rates include simplifying the model, and running the simulation on a faster PC processor. See "Delay and Latency" on page 3-85 of this book, and "Improving Simulation Performance and Accuracy" in the Simulink documentation, for other ideas on improving simulation performance.

Dialog Box



Sample rate (Hz)

The sample rate of the audio data to be acquired. Select one of the standard Windows rates or the **User-defined** option.

User-defined sample rate (Hz)

The (nonstandard) sample rate of the audio data to be acquired.

Sample width (bits)

The number of bits used to represent each signal sample.

Stereo

Specifies stereo (two-channel) inputs when checked, mono (one-channel) inputs when unchecked. Stereo output is M-by-2; mono output is M-by-1.

Samples per frame

The number of audio samples in each successive output frame, M.

Queue duration (seconds)

The length of signal (in seconds) to buffer to the hardware at the start of the simulation.

Use default audio device

Reads audio input from the system's default audio device when selected. Deselect to enable the **Audio device ID** parameter and manually enter a device ID number.

Audio device ID

The number of the audio device from which to read the audio output. In a system with several audio devices installed, a value of `1` selects the first audio card, a value of `2` selects the second audio card, and so on. Select **Use default audio device** if the system has only a single audio card installed.

See Also

From Wave File	DSP Blockset
To Wave Device	DSP Blockset

See “Importing WAV Files” on page 3-71 for related information.

From Wave File

Purpose

Read audio data from a Microsoft Wave (`.wav`) file. (Windows 95/98/NT only)

Library

DSP Sources

Description

The From Wave File block reads audio data from a Microsoft Wave (`.wav`) file and generates a double-precision signal with amplitudes in the range ±1. The audio data must be in uncompressed PCM (pulse code modulation) format.

```
y = wavread('filename')    % Equivalent MATLAB code
```

The **File name** parameter can specify an absolute or relative path to the file. If the file is on the MATLAB path or in the current directory (the directory returned by typing `pwd` at the MATLAB command line), you need only specify the file's name. You do not need to specify the `.wav` extension in either case.

If the audio file contains two channels (stereo), the block's output is an M-by-2 matrix containing one frame (M consecutive samples) of audio data from each of the two channels. If the audio file contains a single channel (mono), the block's output is an M-by-1 matrix containing one frame (M consecutive samples) of mono audio data. The frame size, M, is specified by the **Samples per frame** parameter. For M=1, the output is sample-based; otherwise, the output is frame-based.

The output frame period, T_{fo}, is

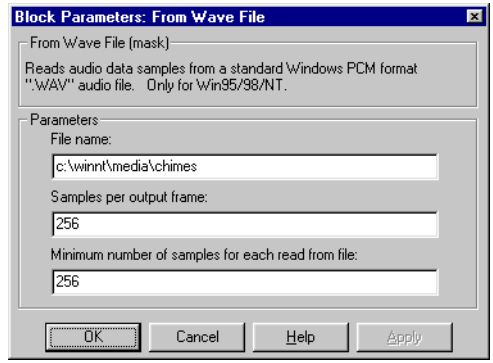
$$T_{fo} = \frac{M}{F_s},$$

where F_s is the data sample rate in Hz.

To reduce the required number of file accesses, the block acquires L consecutive samples from the file during each access, where L is specified by the **Minimum number of samples for each read from file** parameter ($L \geq M$). For $L < M$, the block instead acquires M consecutive samples during each access. Larger values of L result in fewer file accesses, which reduces run-time overhead.

The block icon shows the name, sample rate (in Hz), number of channels (1 or 2), and sample width (in bits) of the data in the specified audio file. All sample rates are supported; the sample width must be either 8 or 16 bits.

Dialog Box



File name

The path and name of the file to read. Paths can be relative or absolute.

Samples per output frame

The number of samples in each output frame, M.

Minimum number of samples for each read from file

The number of consecutive samples to acquire from the file with each file access, L.

See Also

From Wave Device	DSP Blockset
Signal From Workspace	DSP Blockset
To Wave File	DSP Blockset
`wavread`	MATLAB

See “Importing WAV Files” on page 3-71 for related information.

Histogram

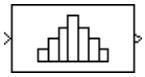
Purpose

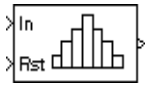
Generate the histogram of an input or sequence of inputs.

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

$$B_m + \left(k + \frac{1}{2}\right)\Delta \qquad k = 0, 1, 2, \ldots, n-1$$

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 *j*th column is the histogram for the data in the *j*th 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.

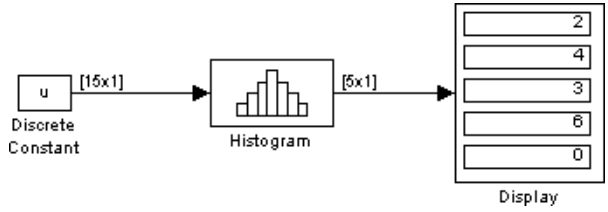
As in basic operation, length-M 1-D vector inputs and *sample-based* length-M row vector inputs are both treated as M-by-1 column vectors.

The output is a sample-based *n*-by-N matrix whose *j*th column reflects the current state of the *j*th histogram. The block resets the running histogram (by emptying all bins of all histograms) when the scalar input at the optional `Rst` port is nonzero. (The `Rst` port can be disabled by deselecting the **Reset port** check box.)

Example

The model below illustrates the Histogram block's basic operation for a single-channel input, `u`, where

```
u = [0 -2 6 -12 2 5 4 3 0 4 3 -2 -3 -2 -9]'
```

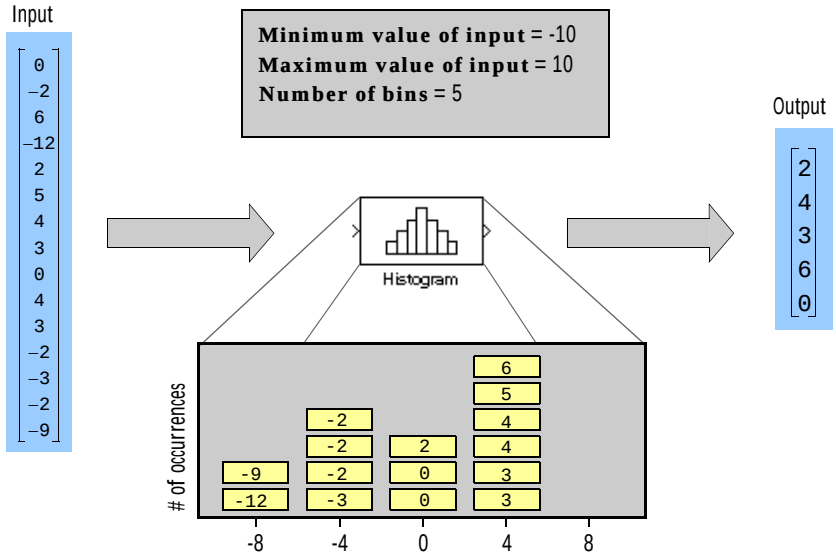


The parameter settings for the Histogram block are:

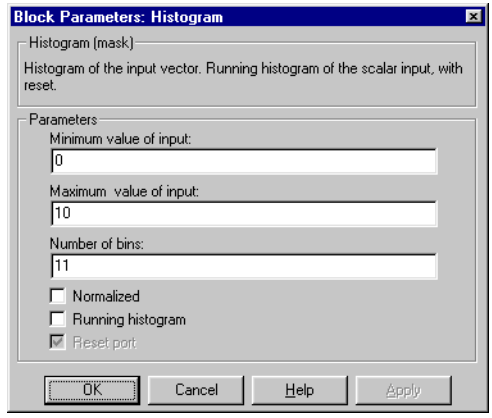
- **Minimum value of input** = `-10`
- **Maximum value of input** = `10`
- **Number of bins** = `5`

- **Normalized** = ☐
- **Running histogram** = ☐

The resulting bin width is 4, as shown below.



Dialog Box



Minimum value of input

The lower boundary, B_m, of the lowest-valued bin.

Maximum value of input

The upper boundary, B_M, of the highest-valued bin.

Number of bins

The number of bins, *n*, in the histogram.

Normalized

Normalizes the output vector (1-norm) when selected. Tunable, except in Simulink's external mode.

Running histogram

Enables running operation when selected.

Reset port

Enables the `Rst` input port when selected.

See Also

Sort	DSP Blockset
`hist`	MATLAB

IDCT

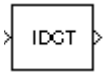
Purpose

Compute the IDCT of the input.

Library

Transforms

Description



The IDCT block computes the inverse discrete cosine transform (IDCT) of each channel in the M-by-N input matrix, u.

```
y = idct(u)        % Equivalent MATLAB code
```

For both sample-based and frame-based inputs, the block assumes that each input column is a frame containing M consecutive samples from an independent channel. The frame size, M, must be a power-of-two. To work with other frame sizes, use the Zero Pad block to pad or truncate the frame size to a power-of-two length.

The output is an M-by-N matrix whose *l*th column contains the length-M IDCT of the corresponding input column.

$$y(m, l) = \sum_{k=1}^{M} w(k)u(k, l)\cos\frac{\pi(2m-1)(k-1)}{2M}, \qquad m = 1, \ldots, M$$

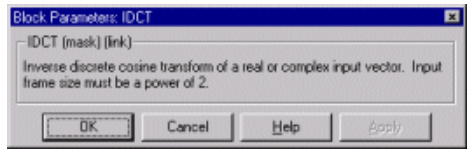
where

$$w(k) = \begin{cases} \frac{1}{\sqrt{M}}, & k = 1 \\ \sqrt{\frac{2}{M}}, & 2 \le k \le M \end{cases}$$

The output is always frame-based, and the output sample rate and data type (real/complex) are the same as those of the input.

For convenience, length-M 1-D vector inputs and *sample-based* length-M row vector inputs are processed as single channels (i.e., as M-by-1 column vectors), and the output has the same dimension as the input.

Dialog Box



See Also

DCT	DSP Blockset
IFFT	DSP Blockset
`idct`	Signal Processing Toolbox

Identity Matrix

Purpose

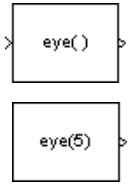
Generate a matrix with ones on the main diagonal and zeros elsewhere.

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 input port attributes** 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
```

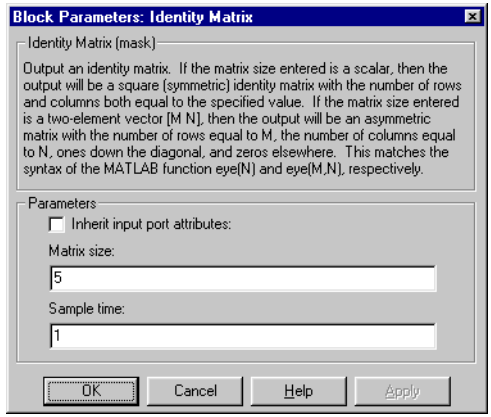
When the **Inherit input port attributes** check box is *not* selected, the input port is disabled, and the dimensions of the output matrix are determined by the **Matrix size** parameter. A scalar value, `M`, specifies an M-by-M identity matrix, while a two-element vector, `[M N]`, specifies an M-by-N unit-diagonal matrix. The output is sample-based, and has the sample period specified by the **Sample time** parameter.

Example

Set **Matrix size** to `[3 6]` to generate the 3-by-6 unit-diagonal matrix below.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Dialog Box



Inherit input port attributes

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** is selected.

Sample time

The discrete sample period of the output. This parameter is disabled when **Inherit input port attributes** is selected.

See Also

Constant Diagonal Matrix	DSP Blockset
DSP Constant	DSP Blockset
eye	MATLAB

See “Creating Signals Using Constant Blocks” on page 3-33 for related information.

IFFT

Purpose

Compute the IFFT of the input.

Library

Transforms

Description

The IFFT block computes the inverse fast Fourier transform (IFFT) of each channel in the complex 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 frequency-samples from an independent channel. The input must be complex, and 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 inverse discrete Fourier transform (IDFT) of the corresponding input column at M evenly spaced time-samples.

$$y(m+1,l) = \frac{1}{M}\sum_{k=0}^{M-1} u(k+1,l)e^{j2\pi(mk/M)} \qquad m = 0,\ldots,M-1$$

The output is always frame-based, and the output port rate is the same as the input port rate. For convenience, length-M 1-D vector inputs and *sample-based* length-M row vector inputs are processed as single channels (i.e., as M-by-1 column vectors), and the output has the same dimension as the input.

When the **Output** parameter is set to **Complex**, the block generates the full complex IDFT.

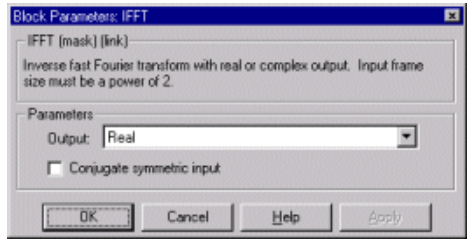
```
y = ifft(u,M)      % Equivalent MATLAB code
```

When the **Output** parameter is set to **Real**, the block generates only the real part of the result.

```
y = real(ifft(u,M))% Equivalent MATLAB code
```

Select the **Conjugate symmetric input** check box if the input to the block is conjugate symmetric. This instructs the block to use an appropriate algorithm and generate a purely real output. A common source of conjugate symmetric data is the FFT block, whose output is conjugate symmetric when the input is purely real.

Dialog Box



Output

The complextiy of the output, **Real** or **Complex**. Specifying **Real** will output only the real part of the IFFT ouput.

Conjugate symmetric input

When selected, specifies that the input is conjugate symmetric.

See Also

FFT	DSP Blockset
IDCT	DSP Blockset
Zero Pad	DSP Blockset
`ifft`	MATLAB

Inherit Complexity

Purpose

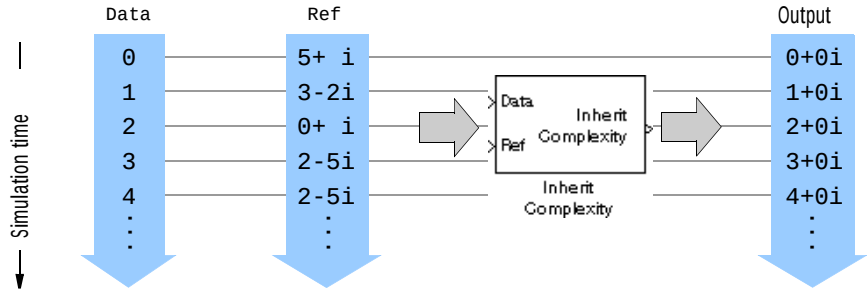
Change the complexity of the input to match that of a reference signal.

Library

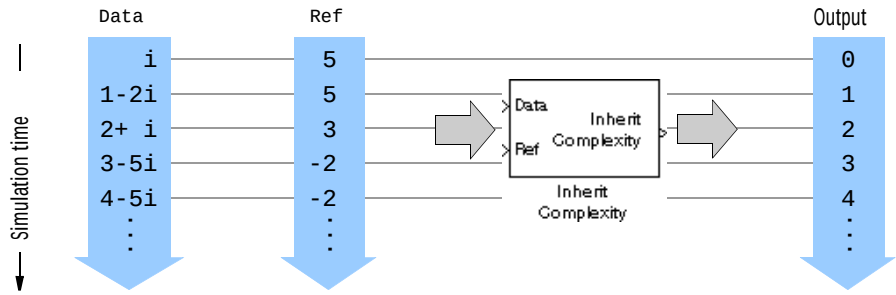
Signal Management / Signal Attributes

Description

The Inherit Complexity block alters the input data at the `Data` port to match the complexity of the reference input at the `Ref` port. If the `Data` input is real, and the `Ref` input is complex, the block appends a zero-valued imaginary component, `0i`, to each element of the `Data` input.

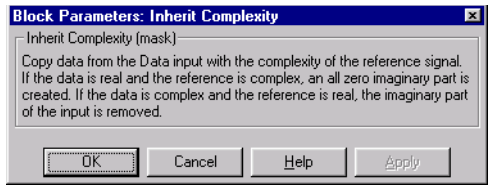


If the `Data` input is complex, and the `Ref` input is real, the block outputs the real component of the `Data` input.



If both the `Data` input and `Ref` input are real, or if both the `Data` input and `Ref` input are complex, the block propagates the `Data` input with no change.

Dialog Box



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

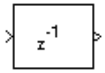
Purpose

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,

the block generates the following sequence of matrices at the start of the simulation,

$$\begin{bmatrix} -1 & -1 \\ -1 & -1 \end{bmatrix}, \begin{bmatrix} u_{11}^{1} & -1 \\ -1 & -1 \end{bmatrix}, \begin{bmatrix} u_{11}^{2} & u_{12}^{1} \\ -1 & -1 \end{bmatrix}, \begin{bmatrix} u_{11}^{3} & u_{12}^{2} \\ u_{21}^{1} & -1 \end{bmatrix}, \begin{bmatrix} u_{11}^{4} & u_{12}^{3} \\ u_{21}^{2} & u_{22}^{1} \end{bmatrix}, \ldots$$

where u_{ij}^{k} is the *i*,*j*th element of the *k*th matrix in the input sequence.

- *Array* of size M-by-N-by-d. In this case, you can set different fixed initial conditions for each element of a sample-based input. This setting is explained further in the *Array* bullet in "Time-Varying Initial Conditions" below.

Initial conditions cannot be specified by full matrices.

Time-Varying Initial Conditions. A time-varying initial condition in sample-based mode can be specified in one of the following ways:

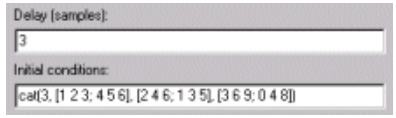
- *Vector* of length d, where d is the maximum value specified for any channel in the **Delay** parameter. The vector can be a L-by-d, 1-by-d, or 1-by-1-by-d. The d elements of the vector are output in sequence, one at each sample time of the initial delay.

  For a scalar input and the parameters shown below,

  the block outputs the sequence `-1, -1, -1, 0, 1,...` at the start of the simulation.

- *Array* of dimension M-by-N-by-d, where d is the value specified for the **Delay** parameter (the *maximum* value if the **Delay** is a vector) and M and N are the number of rows and columns, respectively, in the input matrix. The d *pages*

of the array are output in sequence, one at each sample time of the initial delay. For a 2-by-3 input, and the parameters below,



the block outputs the matrix sequence

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}, \begin{bmatrix} 2 & 4 & 6 \\ 1 & 3 & 5 \end{bmatrix}, \begin{bmatrix} 3 & 6 & 9 \\ 0 & 4 & 8 \end{bmatrix}$$

at the start of the simulation. Note that setting **Initial conditions** to an array with the same matrix for each entry implements *constant* initial conditions; a different constant initial condition for each input matrix element (channel).

Initial conditions cannot be specified by full matrices.

Frame-Based Operation

When the input is a frame-based M-by-N matrix, the block treats each of the N columns as an independent channel, and delays each channel as specified by the **Delay** parameter.

For frame-based inputs, the **Delay** parameter can be a scalar integer by which to equally delay all channels. It can also be a 1-by-N row vector, each element of which serves as the delay for the corresponding channel of the N-channel input. Likewise, it can also be an M-by-1 column vector, each element of which serves as the delay for one of the corresponding M samples for each channel. The **Delay** parameter can be an M-by-N matrix of positive integers as well; in this case, each element of each channel is delayed by the corresponding element in the delay matrix. For instance, if the fifth element of the third column of the delay matrix was 3, then the fifth element of the third channel of the input matrix is always delayed by three sample-time units.

When a channel is delayed for Δ sample-time units, the output sample at time t is the input sample at time $t - \Delta$. If $t - \Delta$ is negative, then the output is the corresponding value specified in the **Initial conditions** parameter.

The **Initial conditions** parameter specifies the output during the initial delay. Both fixed and time-varying initial conditions can be specified. The *initial delay* for a particular channel is the time elapsed from the start of the simulation until the first input in that channel is propagated to the output.

Fixed Initial Conditions. The settings shown below specify *fixed* initial conditions. The value entered in the **Initial conditions** parameter is repeated at the output for each sample time of the initial delay. A fixed initial condition in frame-based mode can be one of the following:

- *Scalar* value to be repeated for all channels of the output at each sample time of the initial delay. For a general M-by-N input with the parameter settings below,

  the first five samples in each of the N channels are zero. Note that if the frame size is larger than the delay, all of these zeros are all included in the first output from the block.

- *Array* of size 1-by-N-by-D. In this case, you can also specify different fixed initial conditions for each channel. See the *Array* bullet in "Time-Varying Initial Conditions" below for details.

Initial conditions cannot be specified by full matrices.

Time-Varying Initial Conditions. The following settings specify *time-varying* initial conditions. For time-varying initial conditions, the values specified in the **Initial conditions** parameter are output in sequence during the initial delay. A time-varying initial condition in frame-based mode can be specified in the following ways:

- *Vector* of length D, where each of the N channels have the same initial conditions sequence specified in the vector. D is defined as follows:

  - When an element of the delay entry is less than the frame size,

    `D = d + 1`

    where *d* is the maximum delay.

- When the all elements of the delay entry are greater than the input frame size,

  `D = d + input frame size - 1`

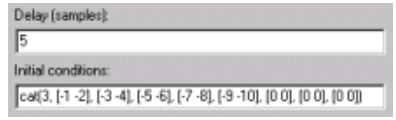
Only the first *d* entries of the initial condition vector will be used; the rest of the values are ignored, but you must include them nonetheless. For a two-channel ramp input `[1:100; 1:100]'` with a frame size of 4 and the parameter settings below,

the block outputs the following sequence of frames at the start of the simulation.

$$\begin{bmatrix} -4 & -1 \\ -5 & -2 \\ 1 & -3 \\ 2 & -4 \end{bmatrix}, \begin{bmatrix} 3 & -5 \\ 4 & 1 \\ 5 & 2 \\ 6 & 3 \end{bmatrix}, \begin{bmatrix} 7 & 4 \\ 8 & 5 \\ 9 & 6 \\ 10 & 7 \end{bmatrix}, \ldots$$

Note that since one of the delays, 2, is less than the frame size of the input, 4, the length of the **Initial conditions** vector is the sum of the maximum delay and 1 (5+1), which is 6. The first five entries of the initial conditions vector are used by the channel with the maximum delay, and the rest of the entries are ignored. Since the first channel is delayed for less than the maximum delay (2 sample time units), it only makes use of two of the initial condition entries.

- *Array* of size 1-by-N-by-D, where D is defined in the *Vector* bullet above in "Time-Varying Initial Conditions" on page 5-211. In this case, the *k*th entry of each 1-by-N entry in the array corresponds to an initial condition for the *k*th channel of the input matrix. Thus, a 1-by-N-by-D initial conditions input allows you to specify different initial conditions for each channel. For instance, for a two-channel ramp input `[1:100; 1:100]'` with a frame size of 4 and the parameter settings below,



the block outputs the following sequence of frames at the start of the simulation.

$$\begin{bmatrix} -1 & -2 \\ -3 & -4 \\ -5 & -6 \\ -7 & -8 \end{bmatrix}, \begin{bmatrix} -9 & -10 \\ 1 & 1 \\ 2 & 2 \\ 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 \\ 5 & 5 \\ 6 & 6 \\ 7 & 7 \end{bmatrix}, \ldots$$

Note that the channels have distinct time varying initial conditions; the initial conditions for channel 1 correspond to the first entry of each length-2 row vector in the initial conditions array, and the initial conditions for channel 2 correspond to the second entry of each row vector in the initial conditions array. Only the first five entries in the initial conditions array are used; the rest are ignored.

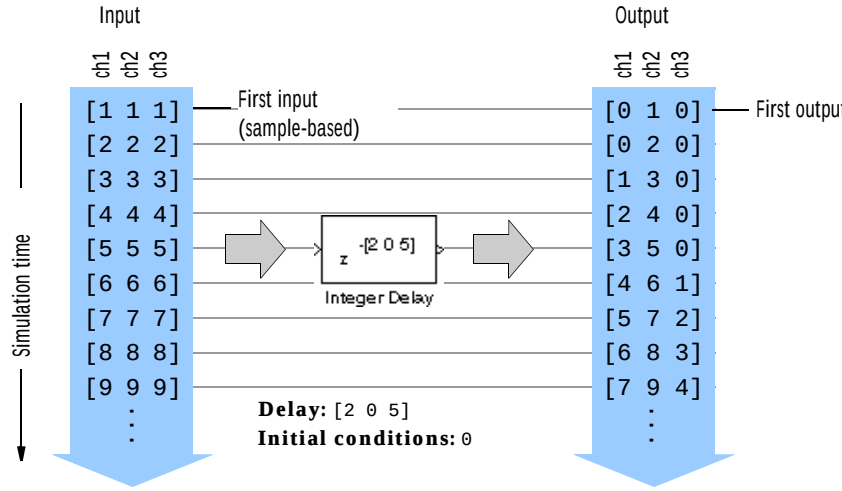
The 1-by-N-by-D array entry can also specify different *fixed* initial conditions for every channel; in this case, every 1-by-N entry in the array would be identical, so that the initial conditions for each column are fixed over time.

Initial conditions cannot be specified by full matrices.

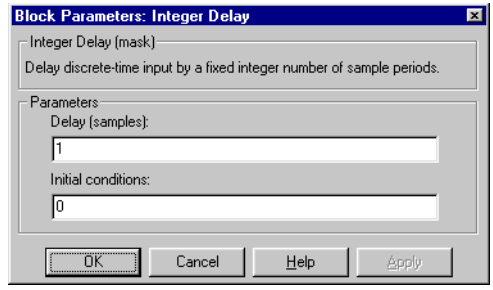
Examples

The `dspafxr` demo illustrates an audio reverberation system built around the Integer Delay block.

Integer Delay



Dialog Box



Delay

The number of sample periods to delay the input signal.

Initial conditions

The value of the block's output during the initial delay.

See Also

Unit Delay	Simulink
Variable Fractional Delay	DSP Blockset
Variable Integer Delay	DSP Blockset

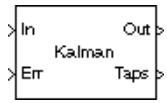
Purpose

Compute filter estimates for an input using the Kalman adaptive filter algorithm.

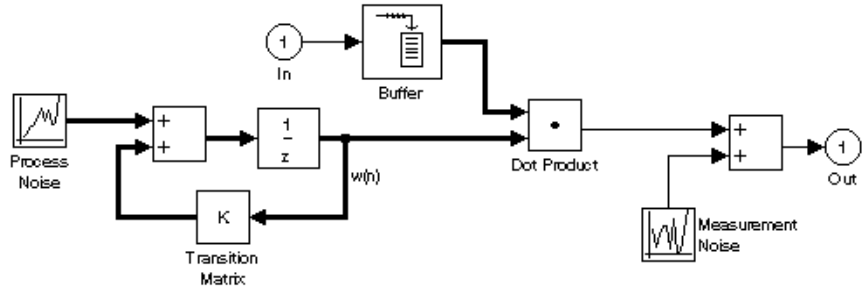
Library

Filtering / Adaptive Filters

Description



The Kalman Adaptive Filter block computes the optimal linear minimum mean-square estimate (MMSE) of the FIR filter coefficients using a one-step predictor algorithm. This Kalman filter algorithm is based on the following physical realization of a dynamical system.



The Kalman filter assumes that there are no deterministic changes to the filter taps over time (i.e., the transition matrix is identity), and that the only observable output from the system is the filter output with additive noise. The corresponding Kalman filter is expressed in matrix form as

$$g(n) = \frac{K(n-1)u(n)}{u^H(n)K(n-1)u(n) + Q_M}$$

$$y(n) = u^H(n)\hat{w}(n)$$

$$e(n) = d(n) - y(n)$$

$$\hat{w}(n+1) = \hat{w}(n) + e(n)g(n)$$

$$K(n) = K(n-1) - g(n)u^H(n)K(n-1) + Q_P$$

The variables are as follows.

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

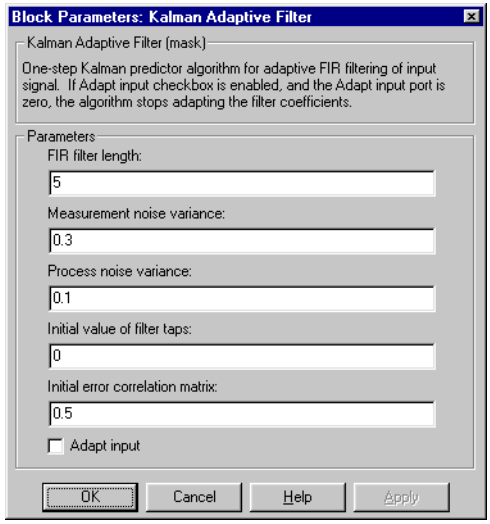
An optional `Adapt` input port is added when the **Adapt input** check box is selected in the dialog box. When this port is enabled, the block continuously adapts the filter coefficients while the `Adapt` input is nonzero. A zero-valued input to the `Adapt` port causes the block to stop adapting, and 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.

Kalman Adaptive Filter

Dialog Box



FIR filter length

The length of the FIR filter.

Measurement noise variance

The value to appear along the diagonal of the measurement noise correlation matrix. Tunable.

Process noise variance

The value to appear along the diagonal of the process noise correlation matrix. Tunable.

Initial value of filter taps

The initial FIR filter coefficients.

Initial error correlation matrix

The initial value of the error correlation matrix.

Adapt input

Enables the `Adapt` port.

References

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

See Also

LMS Adaptive Filter	DSP Blockset
RLS Adaptive Filter	DSP Blockset

See “Adaptive Filters” on page 4-3 for related information.

LDL Factorization

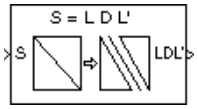
Purpose

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.

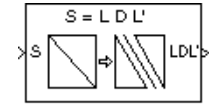
The following options are available:

- **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 ouput.
- **Warning** – Display a warning message in the MATLAB command window, and continue the simulation. The output is *not* a valid factorization. A partial factorization will be present in the upper left corner of the ouput.
- **Error** – Display an error dialog box and terminate the simulation.

Example

LDL decomposition of a 3-by-3 Hermitian positive definite matrix:

$$\begin{bmatrix} 9 & -1 & 2 \\ -1 & 8 & -5 \\ 2 & -5 & 7 \end{bmatrix}$$

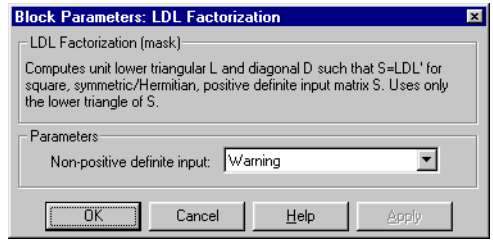


LDL Factorization

$$\begin{bmatrix} 9.00 & -0.11 & 0.22 \\ -0.11 & 7.89 & -0.61 \\ 0.22 & -0.61 & 3.66 \end{bmatrix}$$

$$L = \begin{bmatrix} 1 & 0 & 0 \\ -0.11 & 1 & 0 \\ 0.22 & -0.61 & 1 \end{bmatrix} \quad D = \begin{bmatrix} 9.00 & 0 & 0 \\ 0 & 7.89 & 0 \\ 0 & 0 & 3.66 \end{bmatrix} \quad L' = \begin{bmatrix} 1 & -0.11 & 0.22 \\ 0 & 1 & -0.61 \\ 0 & 0 & 1 \end{bmatrix}$$

Dialog Box



Non-positive definite input

Response to non-positive definite matrix inputs. Tunable.

References

Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

LDL Factorization

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 4-32 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.

$$S^{-1} = (LDL^{*})^{-1}$$

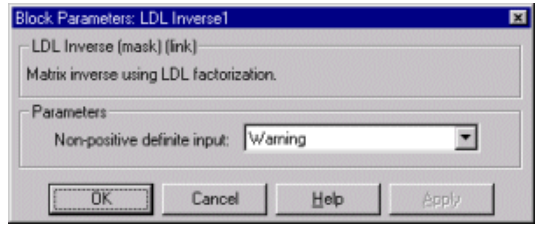
L is a lower triangular square matrix with unity diagonal elements, D is a diagonal matrix, and L^{*} is the Hermitian (complex conjugate) transpose of L. Only the diagonal and lower triangle of the input matrix are used, and any imaginary component of the diagonal entries is disregarded. The output 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



Non-positive definite input

Response to non-positive definite matrix inputs. Tunable.

References

Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

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 4-34 for related information.

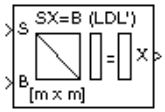
Purpose

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-hand side M-by-N matrix, B. The output is the unique solution of the equations, M-by-N matrix *X*, and is always sample-based.

A length-M 1-D vector input for right-hand side B is treated as an M-by-1 matrix.

When the input is not positive definite, the block reacts with the behavior specified by the **Non-positive definite input** parameter. The following options are available:

- **Ignore** – Proceed with the computation and *do not* issue an alert. The output is *not* a valid solution.
- **Warning** – Proceed with the computation and display a warning message in the MATLAB command window. The output is *not* a valid solution.
- **Error** – Display an error dialog box and terminate the simulation.

Algorithm

The LDL algorithm uniquely factors the Hermitian positive definite input matrix S as

$$S = LDL^*$$

where L is a lower triangular square matrix with unity diagonal elements, D is a diagonal matrix, and L^* is the Hermitian (complex conjugate) transpose of L.

The equation

$$LDL^*X = B$$

is solved for *X* by the following steps:

1 Substitute

$$Y = DL^{*}X$$

2 Substitute

$$Z = L^{*}X$$

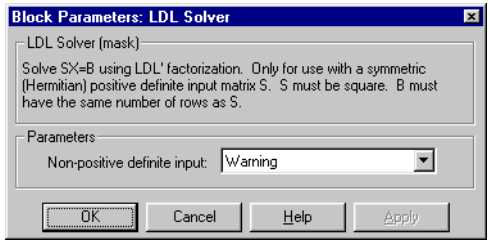
3 Solve one diagonal and two triangular systems.

$$LY = B$$

$$DZ = Y$$

$$L^{*}X = Z$$

Dialog Box



Non-positive definite input

Response to non-positive definite matrix inputs. Tunable.

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 4-31 for related information.

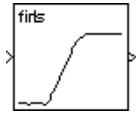
Purpose

Design and implement a least-squares FIR filter.

Library

Filtering / Filter Designs

Description



The Least Squares FIR Filter Design block designs an FIR filter and applies it to a discrete-time input using the Direct-Form II Transpose Filter block. The filter design uses the `firls` function in the Signal Processing Toolbox to minimize the integral of the squared error between the desired frequency response and the actual frequency response.

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

The **Filter type** parameter allows you to specify one of the following filters:

- **Multiband**

  The **Multiband** filter designs a linear-phase filter with an arbitrary magnitude response.

- **Differentiator**

  The **Differentiator** filter approximates the ideal differentiator. Differentiators are antisymmetric FIR filters with approximately linear magnitude responses. To obtain the correct derivative, scale the **Gains at these frequencies** vector by πF_s rad/s, where F_s is the sample frequency in Hertz.

- **Hilbert Transformer**

  The **Hilbert Transformer** filter approximates the ideal Hilbert transformer. Hilbert transformers are antisymmetric FIR filters with approximately constant magnitude.

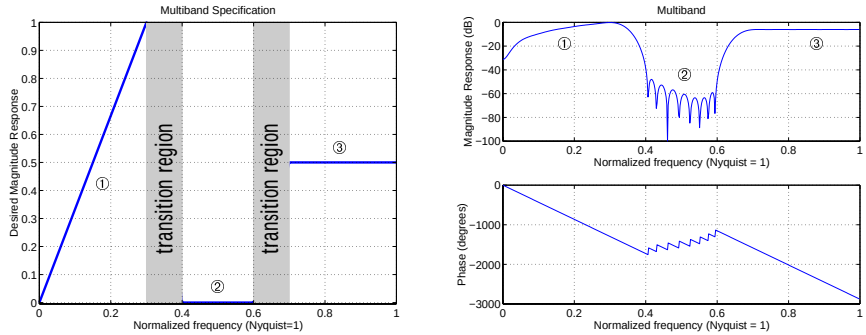
The **Band-edge frequency vector** parameter is a vector of frequency points in the range `0` to `1`, where `1` corresponds to half the sample frequency. This vector must have even length, and intermediate points must appear in ascending order. The **Gains at these frequencies** parameter is a vector containing the desired magnitude response at the corresponding points in the **Band-edge frequency vector**.

Each odd-indexed frequency-amplitude pair defines the left endpoint of a line segment representing the desired magnitude response in that frequency band. The corresponding even-indexed frequency-amplitude pair defines the right endpoint. Between the frequency bands specified by these end-points, there may be undefined sections of the specified frequency response. These are called "don't care" or "transition" regions, and the magnitude response in these areas is a result of the optimization in the other (specified) frequency ranges.

Band edge frequency = `[0 0.3 0.4 0.6 0.7 1]`

Gains = `[0 1 0 0 0.5 0.5]`

Band: ① ② ③



The **Weights** parameter is a vector that specifies the emphasis to be placed on minimizing the error in certain frequency bands relative to others. This vector specifies one weight per band, so it is half the length of the **Band-edge frequency vector** and **Gains at these frequencies** vectors.

In most cases, differentiators and Hilbert transformers have only a single band, so the weight is a scalar value that does not affect the final filter. However, the **Weights** parameter is useful when using the block to design an antisymmetric multiband filter, such as a Hilbert transformer with stopbands.

For more information on the **Band-edge frequency vector**, **Gains at these frequencies**, and **Weights** parameters, see "Filter Designs" on page 4-3. For more on the FIR filter algorithm, see the description of the `firls` function in the Signal Processing Toolbox documentation.

Examples

Example 1: Multiband

Consider a lowpass filter with a transition band in the normalized frequency range 0.4 to 0.5, and 10 times more error minimization in the stopband than the passband. In this case,

- **Filter type** = **Multiband**
- **Band-edge frequency vector** = `[0 0.4 0.5 1]`
- **Gains at these frequencies** = `[1 1 0 0]`
- **Weights** = `[1 10]`

Example 2: Differentiator

Assume the specifications for a differentiator filter require it to have order 21. The "ramp" response extends over the entire frequency range. In this case, specify:

- **Filter type** = **Differentiator**
- **Filter order** = `21`
- **Band-edge frequency vector** = `[0 1]`
- **Gains at these frequencies** = `[0 pi*Fs]`

For a type III (even order) filter, the differentiation band should stop short of half the sample frequency. For example, if the filter order is 20, you could specify the block parameters as follows:

- **Filter type** = **Differentiator**
- **Filter order** = `20`
- **Band-edge frequency vector** = `[0 0.9]`
- **Gains at these frequencies** = `[0 0.9*pi*Fs]`

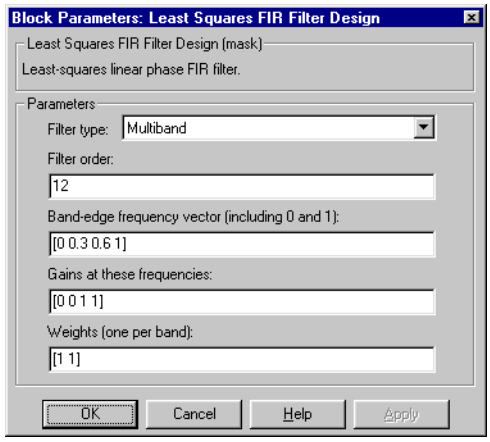
Example 3: Hilbert Transformer

Assume the specifications for a Hilbert transformer filter require it to have order 21. The passband extends over approximately the entire frequency range. In this case, specify:

- **Filter type** = **Hilbert Transform**
- **Filter order** = `21`
- **Band-edge frequency vector** = `[0.1 1]`

- **Gains at these frequencies** = `[1 1]`

Dialog Box



Filter type

The filter type. Tunable.

Filter order

The filter order.

Band-edge frequency vector

A vector of frequency points, in ascending order, in the range `0` to `1`. The value `1` corresponds to half the sample frequency. This vector must have even length. Tunable.

Gains at these frequencies

A vector of frequency-response amplitudes corresponding to the points in the **Band-edge frequency vector**. This vector must be the same length as the **Band-edge frequency vector**. Tunable.

Weights

A vector containing one weight for each frequency band. This vector must be half the length of the **Band-edge frequency vector** and **Gains at these frequencies** vectors. Tunable.

References

Oppenheim, A. V. and R. W. Schafer. *Discrete-Time Signal Processing.* Englewood Cliffs, NJ: Prentice Hall, 1989.

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

See Also

Digital FIR Filter Design	DSP Blockset
Remez FIR Filter Design	DSP Blockset
Yule-Walker IIR Filter Design	DSP Blockset
`firls`	Signal Processing Toolbox

See “Filter Designs” on page 4-3 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 nth order polynomial that best fits the input data in the least-squares sense, where n is specified by the **Polynomial order** parameter. A distinct set of n+1 coefficients is computed for each column of the M-by-N input, u.

For a given input column, the block computes the set of coefficients, $c_1, c_2, \ldots, c_{n+1}$, that minimizes the quantity

$$\sum_{i=1}^{M} (u_i - \hat{u}_i)^2$$

where u_i is the ith element in the input column, and

$$\hat{u}_i = f(x_i) = c_1 x_i^n + c_2 x_i^{n-1} + \cdots + c_{n+1}$$

The values of the independent variable, $x_1, x_2, \ldots, x_M$, are specified as a length-M vector by the **Control points** parameter. The same M control points are used for all N polynomial fits, and can be equally or unequally spaced. The equivalent MATLAB code is shown below.

```
c = polyfit(x,u,n)      % Equivalent MATLAB code
```

Inputs can be frame-based or sample-based. For convenience, a length-M 1-D vector input is treated as an M-by-1 matrix.

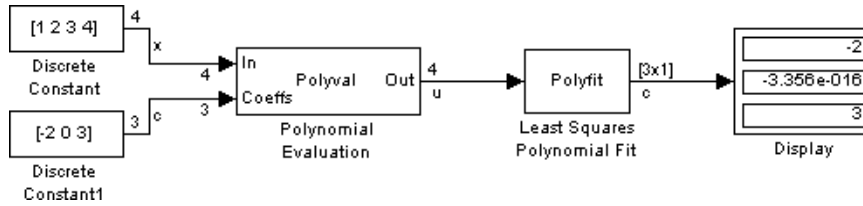
Each column of the (n+1)-by-N output matrix, c, represents a set of n+1 coefficients describing the best-fit polynomial for the corresponding column of the input. The coefficients in each column are arranged in order of descending exponents, $c_1, c_2, \ldots, c_{n+1}$. The output is always sample-based.

Example

In the model below, the Polynomial Evaluation block uses the second-order polynomial

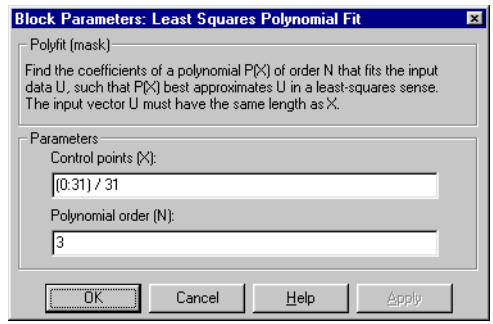
$$y = -2u^2 + 3$$

to generate four values of dependent variable *y* from four values of independent variable *u*, received at the top port. The polynomial coefficients are supplied in the vector `[-2 0 3]` at the bottom port. Note that the coefficient of the first-order term is zero.



The **Control points** parameter of the Least Squares Polynomial Fit block is configured with the same four values of independent variable *u* that are used as input to the Polynomial Evaluation block, `[1 2 3 4]`. The Least Squares Polynomial Fit block uses these values together with the input values of dependent variable *y* to reconstruct the original polynomial coefficients.

Dialog Box



Control points

The values of the independent variable to which the data in each input column correspond. For an M-by-N input, this parameter must be a length-M vector.

Polynomial order

The order, *n*, of the polynomial to be used in constructing the best fit. The number of coefficients is *n*+1.

See Also

Detrend	DSP Blockset
Polynomial Evaluation	DSP Blockset
Polynomial Stability Test	DSP Blockset
`polyfit`	MATLAB

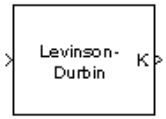
Purpose

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 symmetric, positive-definite, Toeplitz matrix and b is identical to the first column of R shifted by one element and with the opposite sign.

$$\begin{bmatrix} r(1) & r(2) & \cdots & r(n) \\ r(2) & r(1) & \cdots & r(n-1) \\ & & & \\ r(n) & r(n-1) & \cdots & r(1) \end{bmatrix} \begin{bmatrix} a(2) \\ a(3) \\ \\ a(n+1) \end{bmatrix} = \begin{bmatrix} -r(2) \\ -r(3) \\ \\ -r(n+1) \end{bmatrix}$$

The input to the block, `r = [r(1) r(2) ... r(n+1)]`, contains lags 0 through n of an autocorrelation sequence that appear in the matrix R, and can be a 1-D or 2-D vector (row or column).

The **Output(s)** parameter allows you to select between two representations of the solution:

- **A** – The output, `[1 a(2) a(3) ... a(n+1)]`, is the solution to the Levinson-Durbin equation, and 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** – The output, `[k(1) k(2) ... k(n)]`, 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. (A scalar input causes an error when **A and K** is selected.)

The output is always sample-based.

When the **Special-case handling of zero-input** check box is selected (default), an input whose `r(1)` element is zero generates a zero-valued output. When the 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 (or all-pole IIR filter) with assumed 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 R*a* = 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.

$$H(z) = \frac{1}{A(z)} = \frac{1}{1 + a(2)z^{-1} + \ldots + a(n+1)z^{-n}}$$

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*).

$$H(z) = A(z) = 1 + a(2)z^{-1} + \ldots + a(n+1)z^{-n}$$

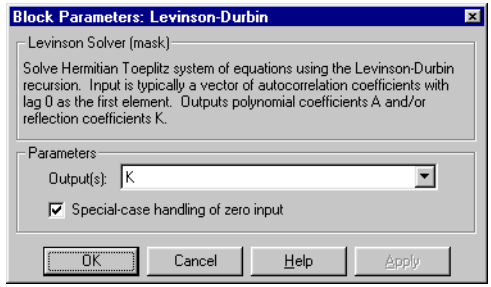
Again, the output at the block's `K` port contains the corresponding reflection coefficients, `[k(1) k(2) ... k(n)]`, for the lattice realization of this FIR

filter. The Autocorrelation LPC block in the Linear Prediction library implements this autocorrelation-based prediction method.

Algorithm

The algorithm requires $O(n^2)$ operations, and is thus much more efficient for large n than standard Gaussian elimination, which requires $O(n^3)$ operations.

Dialog Box



Output(s)

The solution representation of R*a* = b to output: model coefficients (**A**), reflection coefficients (**K**), or both (**A and K**). For scalar inputs, this parameter must be set to **A**.

Special-case handling of zero input

When selected, the block outputs a zero-vector for inputs having `r(1) = 0`. When unselected, 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.

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

Levinson-Durbin

See “Solving Linear Systems” on page 4-31 for related information.

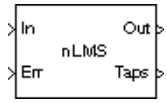
Purpose

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.

$$y(n) = \hat{w}^H(n-1)u(n)$$
$$e(n) = d(n) - y(n)$$
$$\hat{w}(n) = \hat{w}(n-1) + \frac{u(n)}{a + u^H(n)u(n)}\mu e^*(n)$$

The variables are as follows.

Variable	Description
n	The current algorithm iteration
$u(n)$	The buffered input samples at step n
$\hat{w}(n)$	The vector of filter-tap estimates at step n
$y(n)$	The filtered output at step n
$e(n)$	The estimation error at step n
$d(n)$	The desired response at step n
μ	The adaptation step size

To overcome potential numerical instability in the tap-weight update, a small positive constant (a = 1e-10) has been added in the denominator.

To turn off normalization, deselect the **Use normalization** check box in the parameter dialog box. The block then computes the filter-tap estimate as

$$\hat{w}(n) = \hat{w}(n-1) + u(n)\mu e^*(n)$$

The block icon has port labels corresponding to the inputs and outputs of the LMS algorithm. Note that inputs to the `In` and `Err` ports must be sample-based

scalars. The signal at the `Out` port is a scalar, while the signal at the `Taps` port is a sample-based vector.

Block Ports	Corresponding Variables
`In`	u, the scalar input, which is internally buffered into the vector $u(n)$
`Out`	$y(n)$, the filtered scalar output
`Err`	$e(n)$, the scalar estimation error
`Taps`	$\hat{w}(n)$, the vector of filter-tap estimates

An optional `Adapt` input port is added when the **Adapt input** check box is selected in the dialog box. When this port is enabled, the block continuously adapts the filter coefficients while the `Adapt` input is nonzero. A zero-valued input to the `Adapt` port causes the block to stop adapting, and to hold the filter coefficients at their current values until the next nonzero `Adapt` input.

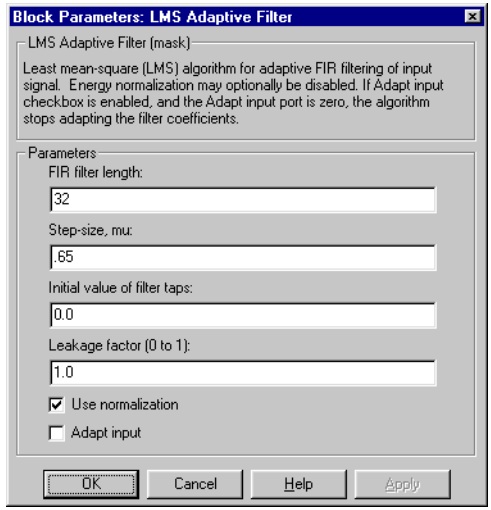
The **FIR filter length** parameter specifies the length of the filter that the LMS algorithm estimates. The **Step size** parameter corresponds to μ in the equations. Typically, for convergence in the mean square, $0<\mu<2$. The **Initial value of filter taps** specifies the initial value $\hat{w}(0)$ as a vector, or as a scalar to be repeated for all vector elements. The **Leakage factor** specifies the value of the leakage factor, $1-\mu\alpha$, in the leaky LMS algorithm below. This parameter must be between 0 and 1.

$$\hat{w}(n+1) = (1-\mu\alpha)\hat{\omega}(n) + \frac{u(n)}{u^{H}(n)u(n)}\mu e^{*}(n)$$

Examples

The `lmsdemo` demo illustrates a noise cancellation system built around the LMS Adaptive Filter block.

Dialog Box



FIR filter length

The length of the FIR filter.

Step-size

The step size, usually in the range (0, 2). Tunable.

Initial value of filter taps

The initial FIR filter coefficients.

Leakage factor

The leakage factor, in the range [0, 1].

Use normalization

Select or deselect normalization.

Adapt input

Enables the `Adapt` port.

References

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

See Also

Kalman Adaptive Filter	DSP Blockset
RLS Adaptive Filter	DSP Blockset

See “Adaptive Filters” on page 4-3 for related information.

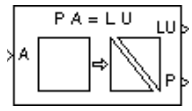
Purpose

Factor a square matrix into lower and upper triangular components.

Library

Math Functions / Matrices and Linear Algebra / Matrix Factorizations

Description



The LU Factorization block factors a row permutation of the square input matrix A as

$$A_p = LU$$

where L is a lower-triangular square matrix with unity diagonal elements, and U is an upper-triangular square matrix. The row-pivoted matrix A_p contains the rows of A permuted as indicated by the permutation index vector P.

```
Ap = A(P,:)          % Equivalent MATLAB code
```

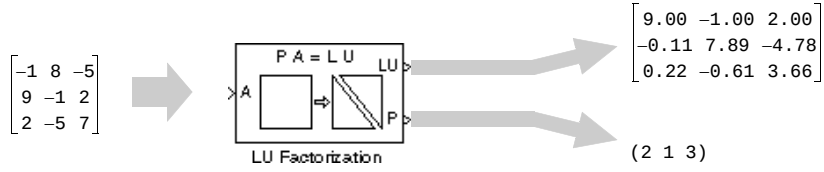
The output at the LU port is a composite matrix with lower subtriangle elements from L and upper triangle elements from U, and is always sample-based.

Example

The row-pivoted matrix A_p and permutation index vector P computed by the block are shown below for 3-by-3 input matrix A.

$$A = \begin{bmatrix} -1 & 8 & -5 \\ 9 & -1 & 2 \\ 2 & -5 & 7 \end{bmatrix} \qquad P = (2\ 1\ 3) \qquad A_p = \begin{bmatrix} 9 & -1 & 2 \\ -1 & 8 & -5 \\ 2 & -5 & 7 \end{bmatrix}$$

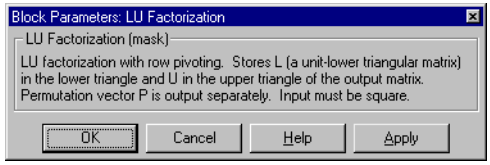
The LU output is a composite matrix whose lower subtriangle forms L and whose upper triangle forms U.



$$L = \begin{bmatrix} 1 & 0 & 0 \\ -0.11 & 1 & 0 \\ 0.22 & -0.61 & 1 \end{bmatrix} \qquad U = \begin{bmatrix} 9.00 & -1.00 & 2.00 \\ 0 & 7.89 & -4.78 \\ 0 & 0 & 3.66 \end{bmatrix}$$

LU Factorization

Dialog Box



References

Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

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 4-32 for related information.

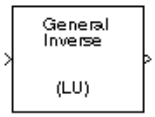
Purpose

Compute the inverse of a square matrix using LU factorization.

Library

Math Functions / Matrices and Linear Algebra / Matrix Inverses

Description

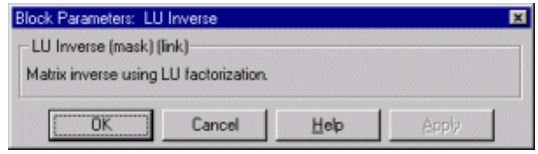


The LU Inverse block computes the inverse of the square input matrix A by factoring and inverting row-pivoted variant A_p.

$$A_p^{-1} = (LU)^{-1}$$

L is a lower-triangular square matrix with unity diagonal elements, and U is an upper-triangular square matrix. The block's output is A^{-1}, and is always sample-based.

Dialog Box



References

Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

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 4-34 for related information.

LU Solver

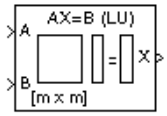
Purpose

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-hand side M-by-N matrix, B. The output is the unique solution of the equations, M-by-N matrix *X*, and is always sample-based.

A length-M 1-D vector input for right-hand side B is treated as an M-by-1 matrix.

Algorithm

The LU algorithm factors a row-permuted variant (A_p) of the square input matrix A as

$$A_p = LU$$

where L is a lower-triangular square matrix with unity diagonal elements, and U is an upper-triangular square matrix.

The matrix factors are substituted for A_p in

$$A_p X = B_p$$

where B_p is the row-permuted variant of B, and the resulting equation

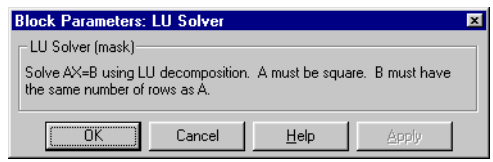
$$LUX = B_p$$

is solved for *X* by making the substitution $Y = UX$, and solving two triangular systems.

$$LY = B_p$$

$$UX = Y$$

Dialog Box



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 4-31 for related information.

Magnitude FFT

Purpose

Compute a nonparametric estimate of the spectrum using the periodogram method.

Library

Estimation / Power Spectrum Estimation

Description

The Magnitude FFT block computes a nonparametric estimate of the spectrum using the periodogram method. For input *u*, this is equivalent to

```
y = abs(fft(u,nfft)).^2    % Equivalent MATLAB code
```

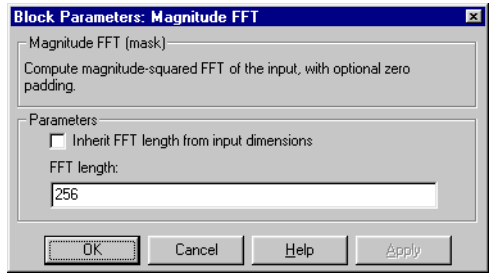
Both an M-by-N frame-based matrix input and an M-by-N sample-based matrix input are treated as M sequential time samples from N independent channels. The block computes a separate estimate for each of the N independent channels and generates an N_{fft}-by-N matrix output. When **Inherit FFT length from 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.

Example

The `dspsacomp` demo compares the periodogram method with several other spectral estimation methods.

Dialog Box



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.

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 4-30 for related information.

Matrix 1-Norm

Purpose

Compute the 1-norm of a matrix.

Library

Math Functions / Matrices and Linear Algebra / Matrix Operations

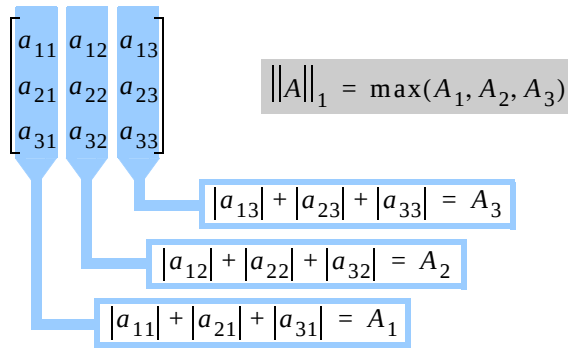
Description

The Matrix 1-Norm block computes the 1-norm, or maximum column-sum, of an M-by-N input matrix, A.

$$y = \|A\|_1 = \max_{1 \le j \le N} \sum_{i=1}^{M} |a_{ij}|$$

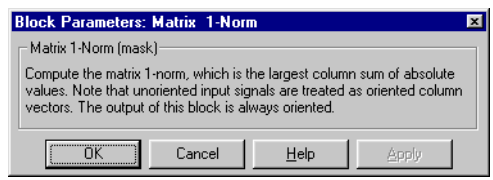
This is equivalent to

```
y = max(sum(abs(A)))   % Equivalent MATLAB code
```



A length-M 1-D vector input is treated as an M-by-1 matrix. The output, *y*, is always a scalar.

Dialog Box



References

Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

See Also

Normalization	DSP Blockset
Reciprocal Condition	DSP Blockset
`norm`	MATLAB

Matrix Multiply

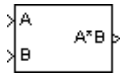
Purpose

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 *n*th port.

```
Y = ((((A*B)*C)*D) ... Un)      % Equivalent MATLAB code
```

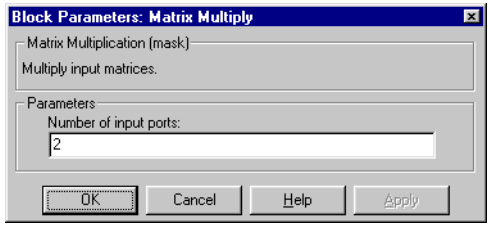
All inputs must have sizes compatible for matrix multiplication; that is, `size(A,2)` = `size(B,1)`, `size(B,2)` = `size(C,1)`, and so on. Inputs can be real, complex, sample-based, or frame-based in any combination, but *all* inputs must have the same precision, single or double. A length-M 1-D vector input at any port is treated as an M-by-1 matrix.

The size of sample-based output `Y` is `[size(A,1) size(Un,2)]`. That is, `Y` is M_A-by-N_{Un}.

Algorithm

The Matrix Multiply block is optimized to use at most two temporary variables for storage of intermediate results.

Dialog Box



Number of input ports

The number of inputs to the block.

See Also

Dot Product	Simulink
Matrix Product	DSP Blockset
Matrix Scaling	DSP Blockset
Product	Simulink

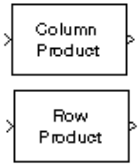
Purpose

Multiply the elements of a matrix along rows or columns.

Library

Math Functions / Matrices and Linear Algebra / Matrix Operations

Description



The Matrix Product block multiplies the elements of an M-by-N input matrix *u* along either the rows or columns.

When the **Multiply along** parameter is set to **Rows**, the block multiplies across the elements of each row and outputs the resulting M-by-1 matrix. A length-N 1-D vector input is treated as a 1-by-N matrix.

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \\ u_{31} & u_{32} & u_{33} \end{bmatrix} \Rightarrow \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} (u_{11}u_{12}u_{13}) \\ (u_{21}u_{22}u_{23}) \\ (u_{31}u_{32}u_{33}) \end{bmatrix}$$

This is equivalent to

```
y = prod(u,2)       % Equivalent MATLAB code
```

When the **Multiply along** parameter is set to **Columns**, the block multiplies down the elements of each column and outputs the resulting 1-by-N matrix. A length-M 1-D vector input is treated as a M-by-1 matrix.

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \\ u_{31} & u_{32} & u_{33} \end{bmatrix}$$

$$\Downarrow$$

$$\begin{bmatrix} y_1 & y_2 & y_3 \end{bmatrix} = \begin{bmatrix} (u_{11}u_{21}u_{31}) & (u_{12}u_{22}u_{32}) & (u_{13}u_{23}u_{33}) \end{bmatrix}$$

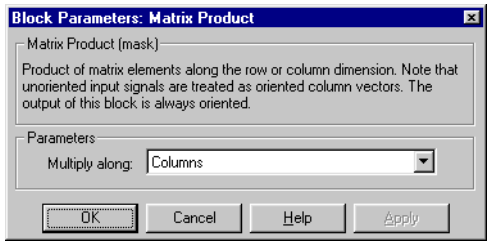
This is equivalent to

```
y = prod(u)         % Equivalent MATLAB code
```

The output has the same frame status as the input.

Matrix Product

Dialog Box



Multiply along

The dimension of the matrix along which to multiply, row or column.

See Also

Matrix Multiply	DSP Blockset
Matrix Square	DSP Blockset
Matrix Sum	DSP Blockset
`prod`	MATLAB

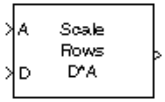
Purpose

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)**, input `D` can be a 1-D or 2-D vector of length M, and the block multiplies each element of `D` across the corresponding *row* of matrix `A`.

$$\begin{bmatrix} d_1 \\ d_2 \\ d_3 \end{bmatrix} \times \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \Rightarrow \begin{bmatrix} d_1 a_{11} & d_1 a_{12} & d_1 a_{13} \\ d_2 a_{21} & d_2 a_{22} & d_2 a_{23} \\ d_3 a_{31} & d_3 a_{32} & d_3 a_{33} \end{bmatrix}$$

This is equivalent to premultiplying `A` by a diagonal matrix with diagonal `D`.

```
y = diag(D)*A       % Equivalent MATLAB code
```

When the **Mode** parameter is set to **Scale Columns (A*D)**, input `D` can be a 1-D or 2-D vector of length N, and the block multiplies each element of `D` across the corresponding *column* of matrix `A`.

$$\begin{bmatrix} d_1 & d_2 & d_3 \end{bmatrix} \times \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \Rightarrow \begin{bmatrix} d_1 a_{11} & d_2 a_{12} & d_3 a_{13} \\ d_1 a_{21} & d_2 a_{22} & d_3 a_{23} \\ d_1 a_{31} & d_2 a_{32} & d_3 a_{33} \end{bmatrix}$$

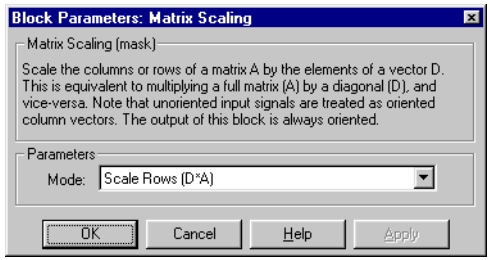
This is equivalent to postmultiplying `A` by a diagonal matrix with diagonal `D`.

```
y = A*diag(D)       % Equivalent MATLAB code
```

The output is the same size as the input matrix, `A`. If both inputs are sample-based, the output is sample-based; otherwise, the output is frame-based.

Matrix Scaling

Dialog Box



Mode

The mode of operation, row scaling or column scaling.

See Also

Matrix Multiply	DSP Blockset
Matrix Product	DSP Blockset
Matrix Sum	DSP Blockset

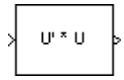
Purpose

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
```

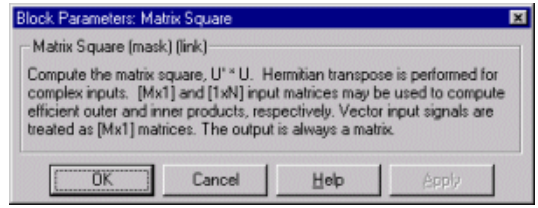
A length-M 1-D vector input is treated as an M-by-1 matrix. For both sample-based and frame-based inputs, output y is sample-based with dimension N-by-N.

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



See Also

Matrix Multiply	DSP Blockset
Matrix Product	DSP Blockset
Matrix Sum	DSP Blockset
Transpose	DSP Blockset

Matrix Sum

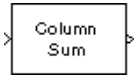
Purpose

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.

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \\ u_{31} & u_{32} & u_{33} \end{bmatrix}$$

$$\Downarrow$$

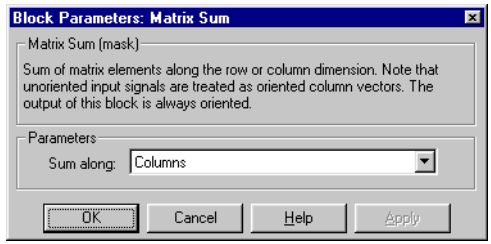
$$\begin{bmatrix} y_1 & y_2 & y_3 \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{3} u_{i1} & \sum_{i=1}^{3} u_{i2} & \sum_{i=1}^{3} u_{i3} \end{bmatrix}$$

This is equivalent to

```
y = sum(u)          % Equivalent MATLAB code
```

The output has the same frame status as the input.

Dialog Box



Sum along

The dimension of the matrix to sum along, row or column.

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

Click on the **Image properties** check box to expose the image property parameters, which control the colormap and display.

The mapping of matrix element values to colors is specified by the **Colormap matrix**, **Minimum input**, and **Maximum input** parameters. For a colormap with L colors, the colormap matrix has dimension L-by-3, with one row for each color and one column for each element of the RGB triple that defines the color. Examples of RGB triples are

```
[ 1   0   0 ]   (red)
[ 0   0   1 ]   (blue)
[0.8 0.8 0.8]   (light gray)
```

See `ColorSpec` in the MATLAB documentation for complete information about defining RGB triples.

MATLAB provides a number of functions for generating predefined colormaps, such as `hot`, `cool`, `bone`, and `autumn`. Each of these functions accepts the colormap size as an argument, and can be used in the **Colormap matrix** parameter. For example, if you specify `gray(128)` for the **Colormap matrix** parameter, the matrix is displayed in 128 shades of gray. The color in the first row of the colormap matrix is used to represent the value specified by the **Minimum input** parameter, and the color in the last row is used to represent the value specified by the **Maximum input** parameter. Values between the minimum and maximum are quantized and mapped to the intermediate rows of the colormap matrix.

The documentation for MATLAB's `colormap` function provides complete information about specifying colormap matrices, and includes a complete list of the available colormap functions.

Axis Properties

Click on the **Axis properties** check box to expose the axis property parameters, which control labeling and positioning.

The **Axis origin** parameter determines where the first element of the input matrix, `U(1,1)`, is displayed. When **Upper left corner** is specified, the matrix is displayed in *matrix orientation*, with `U(1,1)` in the upper-left corner.

$$\begin{bmatrix} U_{11} & U_{12} & U_{13} & U_{14} \\ U_{21} & U_{22} & U_{23} & U_{24} \\ U_{31} & U_{32} & U_{33} & U_{34} \\ U_{41} & U_{42} & U_{43} & U_{44} \end{bmatrix}$$

When **Lower left corner** is specified, the matrix is flipped vertically to *image orientation*, with `U(1,1)` in the lower-left corner.

$$\begin{bmatrix} U_{41} & U_{42} & U_{43} & U_{44} \\ U_{31} & U_{32} & U_{33} & U_{34} \\ U_{21} & U_{22} & U_{23} & U_{24} \\ U_{11} & U_{12} & U_{13} & U_{14} \end{bmatrix}$$

Axis zoom, when selected, causes the image display to completely fill the figure window. Menus and axis titles are not displayed. This option can also be selected from the right-click pop-up menu in the figure window.

When **Axis zoom** is deselected, 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 deselected, the axis labels and titles are displayed in a gray border surrounding the scope axes, and the window's menus (including **Axes**) and toolbar are visible. This option can also be set in the **Axis properties** panel of the parameter dialog box.
- **Colorbar**, when selected, displays a bar with the specified colormap to the right of the image axes.
- **Save Position** automatically updates the **Figure position** parameter in the **Axis properties** field to reflect the figure window's current position and size on the screen. To make the scope window open at a particular location on the screen when the simulation runs, simply drag the window to the desired location, resize it as needed, and select **Save Position**. Note that the parameter dialog box must be closed when you select **Save Position** in order for the **Figure position** parameter to be updated.

Many of these options can also be accessed by right-clicking with the mouse anywhere on the displayed image. The right-click menu is very helpful when the scope is in zoomed mode and the **Axes** menu is not visible.

Examples

See the demo `dspstfft.mdl` for an example of using the Matrix Viewer block to create a moving spectrogram (time-frequency plot) of a speech signal by updating just one column of the input matrix at each sample time.

Dialog Box

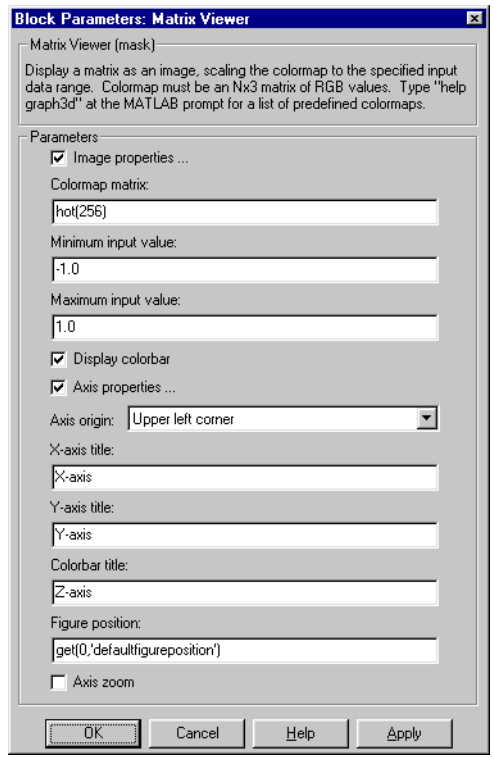


Image properties

Select to expose the image property parameters. Tunable.

Colormap matrix

A 3-column matrix defining the colormap as a set of RGB triples, or a call to a colormap-generating function such as `hot` or `spring`. See the `ColorSpec` property for complete information about defining RGB triples, and the `colormap` function for a list of colormap-generating functions. Tunable.

Minimum input value

The input value to be mapped to the color defined in the first row of the colormap matrix. Select **Autoscale** from the right-click pop-up menu to set this parameter to the minimum value observed in a series of 10 consecutive matrix inputs. Tunable.

Maximum input value

The input value to be mapped to the color defined in the last row of the colormap matrix. Select **Autoscale** from the right-click pop-up menu to set this parameter to the maximum value observed in a series of 10 consecutive matrix inputs. Tunable.

Display colorbar

Select to display a bar with the selected colormap to the right of the image axes. Tunable.

Axis properties

Select to expose the axis property parameters. Tunable.

Axis origin

The position within the axes where the first element of the input matrix, `U(1,1)`, is plotted; bottom left or top left. Tunable.

X-axis title

The text to be displayed below the *x*-axis. Tunable.

Y-axis title

The text to be displayed to the left of the *y*-axis. Tunable.

Colorbar title

The text to be displayed to the right of the color bar, if **Display colorbar** is currently selected. Tunable.

Figure position

A 4-element vector of the form `[left bottom width height]` specifying the position of the figure window, where `(0,0)` is the lower-left corner of the display. Tunable.

Axis zoom

Resizes the image to fill the figure window. Tunable.

See Also

Spectrum Scope	DSP Blockset
Vector Scope	DSP Blockset
`colormap`	MATLAB
`ColorSpec`	MATLAB
`image`	MATLAB

See “Viewing Signals” on page 3-80 for related information.

Maximum

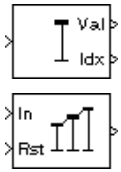
Purpose

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**.

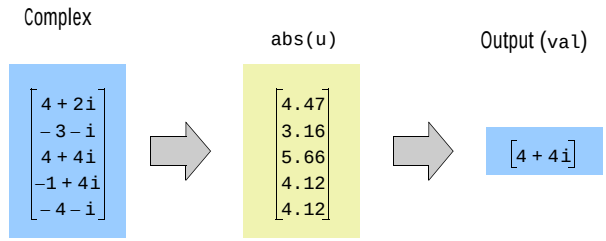
Value Mode

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`.

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 minima, `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.

The block resets the running maximum whenever a reset event is detected at the optional `Rst` port. The reset event is specified by the **Reset port** menu, and can be one of the following:

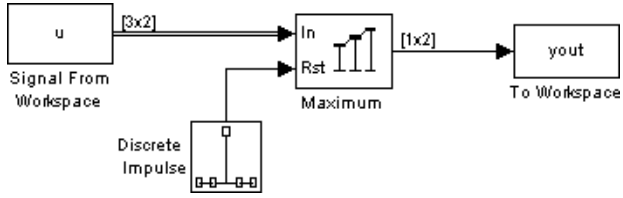
- **Rising edge** triggers a reset operation when the `Rst` input rises from a negative value to zero or a positive value, or from zero to a positive value.
- **Falling edge** triggers a reset operation when the `Rst` input falls from a positive value to zero or a negative value, or from zero to a negative value.
- **Either edge** triggers a reset operation when either a rising or falling edge (as described above) occurs.
- **Non-zero sample** triggers a reset operation at each sample time that the `Rst` input is not zero.

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 `Rst` port can be disabled by selecting **None** from the **Reset port** menu.

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.

Example

The Maximum block in the model below calculates the running maximum of a frame-based 3-by-2 (two-channel) matrix input, u. The running maximum is reset at *t*=2 by an impulse to the block's `Rst` port.



The Maximum block has the following settings:

- **Mode** = **Running**
- **Reset port** = **Non-zero signal**

The Signal From Workspace block has the following settings:

- **Signal** = `u`
- **Sample time** = `1/3`
- **Samples per frame** = `3`

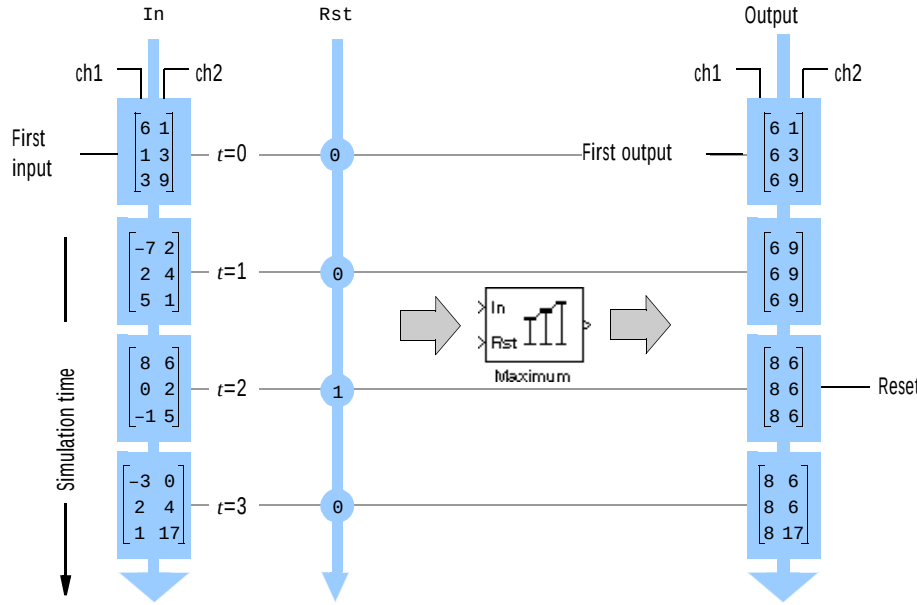
where

```
u = [6 1 3 -7 2 5 8 0 -1 -3 2 1;1 3 9 2 4 1 6 2 5 0 4 17]'
```

The Discrete Impulse block has the following settings:

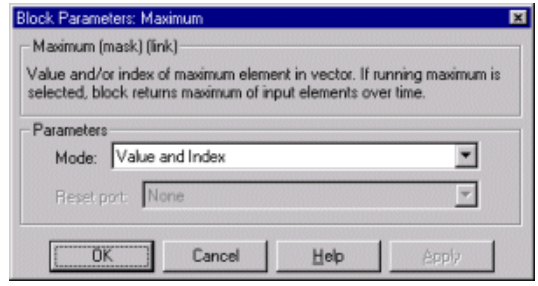
- **Delay (samples)** = `2`
- **Sample time** = `1`
- **Samples per frame** = `1`

The block's operation is shown in the figure below.



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

Dialog Box



Mode

The block's mode of operation: Output the maximum value of each input (**Value**), the index of the maximum value (**Index**), both the value and the

index (**Value and index**), or track the maximum value of the input sequence over time (**Running**).

Reset port

Specifies the reset event detected at the `Rst` input port when **Running** is selected as the **Mode** parameter. The reset operation can be set to occur when a rising and/or falling edge is detected at the `Rst` port, (**Rising edge**, **Falling edge**, **Either Edge**), or when a non-zero sample is detected at the `Rst` port (**Non-zero sample**). The `Rst` port can be disabled by selecting **None**.

See Also

Mean	DSP Blockset
Minimum	DSP Blockset
MinMax	Simulink
`max`	MATLAB

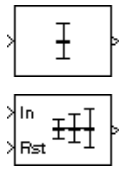
Purpose

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.

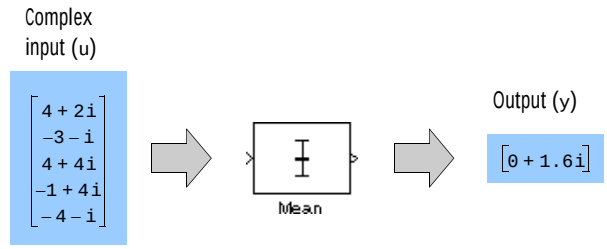
Basic Operation

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

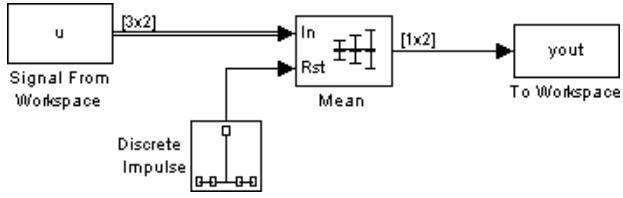
element y_{ij} containing the mean value of the jth column over all inputs since the last reset, up to and including element u_{ij} of the current input.

If the **Reset port** parameter is set to **Non-zero sample**, the optional `Rst` port is enabled and the block resets the running mean when the scalar input at the `Rst` port is nonzero. (The `Rst` port can be disabled by setting the **Reset port** parameter to **None**.) 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.

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.

Example

The Mean block in the model below calculates the running mean of a frame-based 3-by-2 (two-channel) matrix input, u. The running mean is reset at t=2 by an impulse to the block's `Rst` port.



The Mean block has the following settings:

- **Running mean** = ☑
- **Reset port** = **Non-zero sample**

The Signal From Workspace block has the following settings:

- **Signal** = u
- **Sample time** = 1/3
- **Samples per frame** = 3

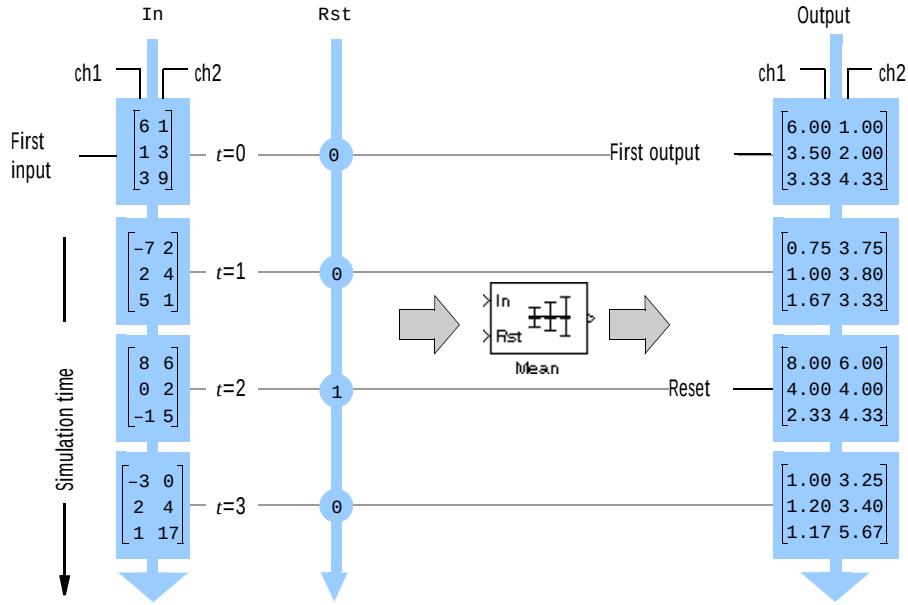
where

```
u = [6 1 3 -7 2 5 8 0 -1 -3 2 1;1 3 9 2 4 1 6 2 5 0 4 17]'
```

The Discrete Impulse block has the following settings:

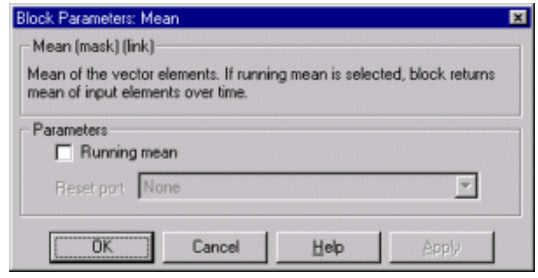
- **Delay (samples)** = 2
- **Sample time** = 1
- **Samples per frame** = 1

The block's operation is shown in the figure below.



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

Dialog Box



Running mean

Enables running operation when selected.

Reset port

Enables the `Rst` input port when set to Non-zero sample, and disables the `Rst` input port when set to None.

See Also

Maximum	DSP Blockset
Median	DSP Blockset
Minimum	DSP Blockset
Standard Deviation	DSP Blockset
`mean`	MATLAB

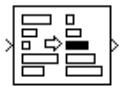
Purpose

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.

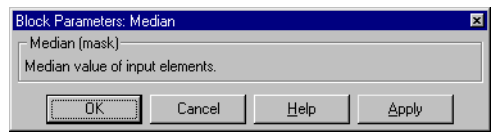
```
s = sort(u);
y = s((M+1)/2,:)
```

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



Median

See Also

Maximum	DSP Blockset
Mean	DSP Blockset
Minimum	DSP Blockset
Sort	DSP Blockset
Standard Deviation	DSP Blockset
Variance	DSP Blockset
`median`	MATLAB

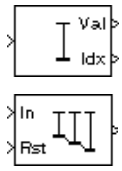
Purpose

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**.

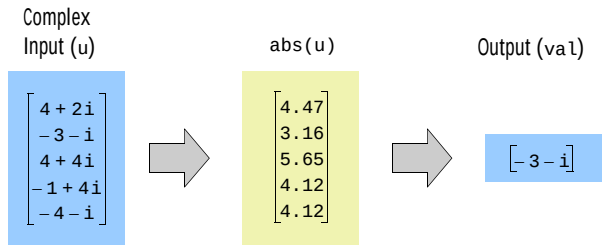
Value Mode

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`.

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

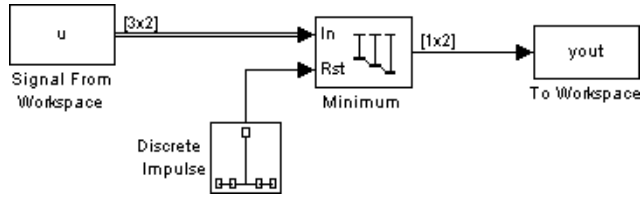
When **Mode** is set to **Running**, the block tracks the minimum value of each channel in a *time-sequence* of M-by-N inputs. For sample-based inputs, the output is a sample-based M-by-N matrix with each element y_{ij} containing the minimum value observed in element u_{ij} for all inputs since the 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.

If the **Reset port** parameter is set to **Non-zero sample**, the optional `Rst` port is enabled and the block resets the running minimum when the scalar input at the `Rst` port is nonzero. (The `Rst` port can be disabled by setting the **Reset port** parameter to **None**.) 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.

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.

Example

The Minimum block in the model below calculates the running minimum of a frame-based 3-by-2 (two-channel) matrix input. The running minimum is reset at t=2 by an impulse to the block's `Rst` port.



The Minimum block has the following settings:

- **Mode** = **Running**
- **Reset port** = **Non-zero sample**

The Signal From Workspace block has the following settings:

- **Signal** = `u`
- **Sample time** = `1/3`
- **Samples per frame** = `3`

where

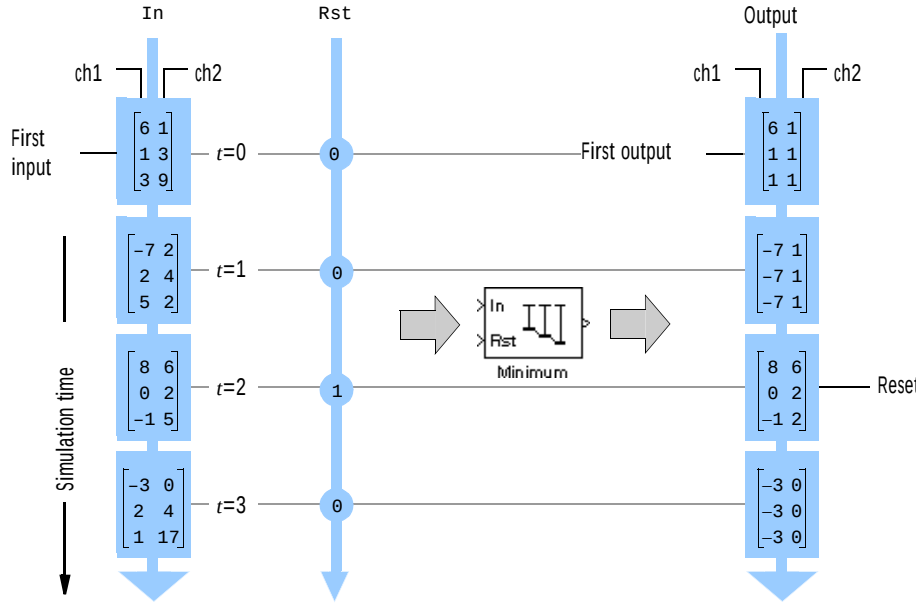
```
u = [6 1 3 -7 2 5 8 0 -1 -3 2 1;1 3 9 2 4 2 6 2 5 0 4 17]'
```

The Discrete Impulse block has the following settings:

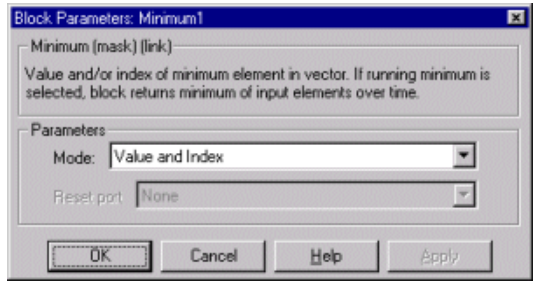
- **Delay (samples)** = `2`
- **Sample time** = `1`
- **Samples per frame** = `1`

The block's operation is shown in the figure below.

Minimum



Dialog Box



Mode

The block's mode of operation: Output the minimum value of each input (**Value**), the index of the minimum value (**Index**), both the value and the index (**Value and Index**), or track the minimum values in the input sequence over time (**Running**).

Reset port

Enables the `Rst` input port when set to **Non-zero sample**, and disables the `Rst` input port when set to **None**.

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

The Modified Covariance AR Estimator block uses the modified covariance method to fit an autoregressive (AR) model to the input data. This method minimizes the forward and backward prediction errors in the least-squares sense. The input is a frame of consecutive time samples, which is assumed to be the output of an AR system driven by white noise. The block computes the normalized estimate of the AR system parameters, $A(z)$, independently for each successive input.

$$H(z) = \frac{G}{A(z)} = \frac{G}{1 + a(2)z^{-1} + \ldots + a(p+1)z^{-p}}$$

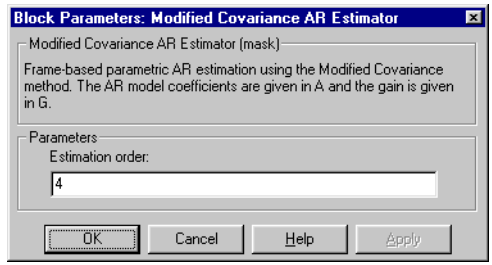
The order, p, of the all-pole model is specified by the **Order** parameter.

The top output, `A`, contains the normalized estimate of the AR model coefficients in descending powers of z,

```
[1 a(2) ... a(p+1)]
```

The scalar gain, G, is provided at the bottom output (`G`).

Dialog Box



Estimation order

The order of the AR model, p.

References

Kay, S. M. *Modern Spectral Estimation: Theory and Application*. Englewood Cliffs, NJ: Prentice-Hall, 1988.

Marple, S. L., Jr., *Digital Spectral Analysis with Applications.* Englewood Cliffs, NJ: Prentice-Hall, 1987.

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

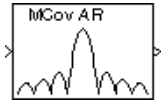
Purpose

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.

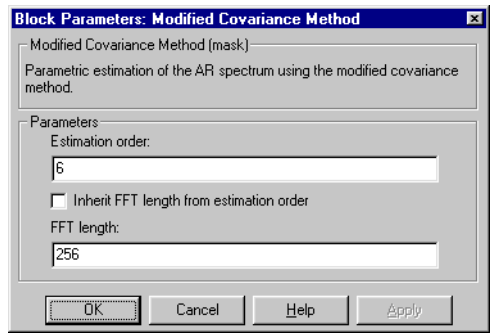
When **Inherit FFT length from input dimensions** is selected, N_{fft} is specified by the frame size of the input, which must be a power of 2. When **Inherit FFT length from input dimensions** is *not* selected, N_{fft} is specified as a power of 2 by the **FFT length** parameter, and the block zero 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



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.

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.

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 4-30 for related information.

Multiphase Clock

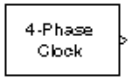
Purpose

Generate multiple binary clock signals.

Library

Signal Management / Switches and Counters

Description



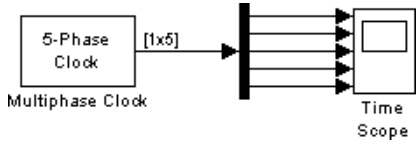
The Multiphase Clock block generates a sample-based 1-by-N vector of clock signals, where the integer N is specified by the **Number of phases** parameter. Each of the N phases has the same frequency, *f*, specified in hertz by the **Clock frequency** parameter.

The clock signal indexed by the **Starting phase** parameter is the first to become active, at *t*=0. The other signals in the output vector become active in turn, each one lagging the preceding signal's activation by 1/(N∗*f*) seconds, the *phase interval*. The period of the sample-based output is therefore 1/(N∗*f*) seconds.

The *active level* can be either high (`1`) or low (`0`), as specified by the **Active level (polarity)** parameter. The duration of the active level, D, is set by the **Number of phase intervals over which the clock is active**. This value, which can be an integer value between 1 and N-1, specifies the number of phase intervals that each signal should remain in the active state after becoming active. The *active duty cycle* of the signal is D/N.

Example

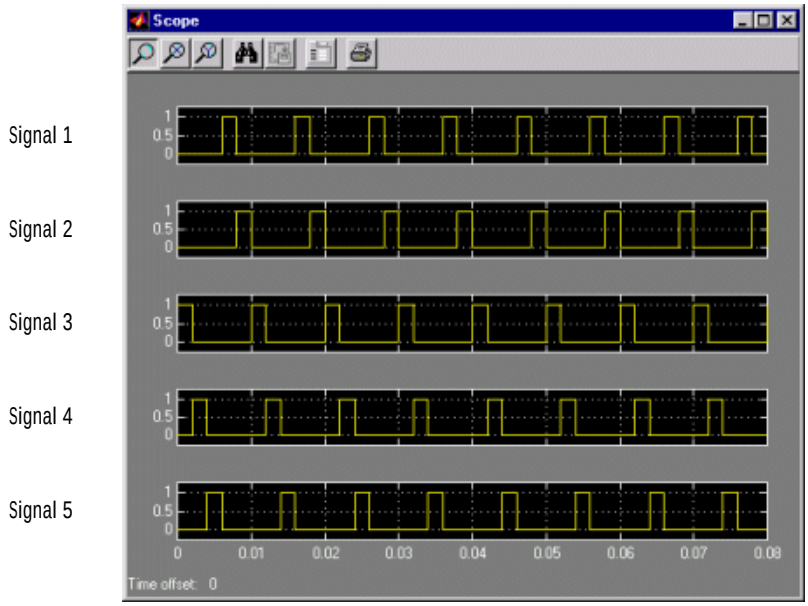
Configure the Multiphase Clock block in the model below to generate a 100 Hz five-phase output in which the third signal is first to become active. Use a *high* active level with a duration of one interval.



The corresponding settings are as follows:

- **Clock frequency** = `100`
- **Number of phases** = `5`
- **Starting phase** = `3`
- **Number of phase intervals over which the clock is active** = `1`
- **Active level (polarity)** = **High (1)**

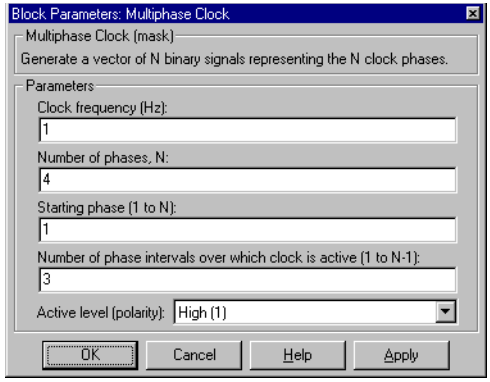
The Scope window below shows the Multiphase Clock block's output for these settings. Note that the first active level appears at t=0 on `y(3)`, the second active level appears at t=0.002 on `y(4)`, the third active level appears at t=0.004 on `y(5)`, the fourth active level appears at t=0.006 on `y(1)`, and the fifth active level appears at t=0.008 on `y(2)`. Each signal becomes active 1/(5∗100) seconds after the previous signal.



To experiment further, try changing the **Number of phase intervals over which clock is active** setting to `3` so that the active-level duration is three phase intervals (60% duty cycle).

Multiphase Clock

Dialog Box



Clock frequency

The frequency of all output clock signals.

Number of phases

The number of different phases, N, in the output vector.

Starting phase

The vector index of the output signal to first become active. Tunable.

Number of phase intervals over which clock is active

The duration of the active level for every output signal. Tunable.

Active level

The active level, high (`1`) or low (`0`). Tunable.

See Also

Clock	Simulink
Counter	DSP Blockset
Discrete Pulse Generator	Simulink
Event-Count Comparator	DSP Blockset

See "Creating Signals Using Signal Generator Blocks" on page 3-36 for related information.

Purpose

Distribute arbitrary subsets of input rows or columns to multiple output ports.

Library

Signal Management / Indexing

Description

The Multi-port Selector block extracts multiple subsets of rows or columns from M-by-N input matrix *u*, and propagates each new submatrix to a distinct output port. A length-M 1-D vector input is treated as an M-by-1 matrix.

The **Indices to output** parameter is a cell array whose *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.

The **Indices to output** parameter is tunable, so you can change the values of the indices at any time during the simulation; however, the number of cells in the array (i.e., the number of output ports) and the size of each submatrix in the output must remain the same while the simulation is running.

When an index references a non-existent row or column of the input, the block reacts with the behavior specified by the **Invalid index** parameter. The following options are available:

- **Clip index** – Clip the index to the nearest valid value, and *do not* issue an alert.

  Example: For a 64-by-4 input with **Select** = **Rows**, an index of 72 is clipped to 64; with **Select** = **Columns**, an index of 72 is clipped to 4. In both cases, an index of -2 is clipped to 1.
- **Clip and warn** – Display a warning message in the MATLAB command window, and clip as above.
- **Generate error** – Display an error dialog box and terminate the simulation.

Example

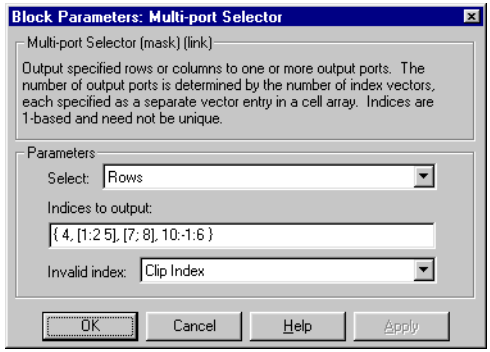
Consider the following **Indices to output** cell array:

```
{4,[1:2 5],[7;8],10:-1:6}
```

This is a four-cell array, which requires the block to generate four independent outputs (each at a distinct port). The table below shows the dimensions of these outputs when **Select** = **Rows** and the input dimension is M-by-N.

Cell	Expression	Description	Output size
1	`4`	Row 4 of input	1-by-N
2	`[1:2 5]`	Rows 1, 2, and 5 of input	3-by-N
3	`[7;8]`	Rows 7 and 8 of input	2-by-N
4	`10:-1:6`	Rows 10, 9, 8, 7, and 6 of input	5-by-N

Dialog Box



Select

The dimension of the input to select, **Rows** or **Columns**.

Indices to output

A cell array specifying the row- or column-subsets to propagate to each of the output ports. The number of cells in the array determines the number of output ports on the block. This parameter is tunable, but the size of the cell array (i.e., the number of output ports) and the size of each submatrix in the output must remain the same while the simulation is running.

Invalid index

Response to an invalid index value. Tunable.

See Also

Permute Matrix	DSP Blockset
Selector	Simulink
Submatrix	DSP Blockset
Variable Selector	DSP Blockset

N-Sample Enable

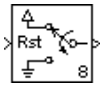
Purpose

Output ones or zeros for a specified number of sample times.

Library

DSP Sources,
Signal Management / Switches and Counters

Description



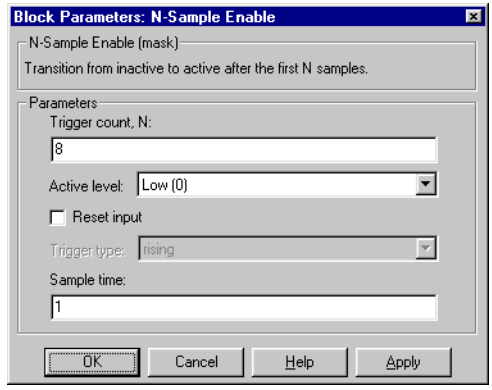
The N-Sample Enable block outputs the *inactive* value (`0` or `1`, whichever is *not* selected in the **Active level** parameter) during the first N sample times, where N is the **Trigger count** value. Beginning with output sample N+1, the block outputs the *active* value (`1` or `0`, whichever is selected in the **Active level** parameter) until a reset event occurs or the simulation terminates.

The **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 triggering event is specified by the **Trigger type** pop-up menu, and can be one of the following:

- **Rising edge** triggers the reset 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 the reset 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 the reset when either a rising or falling edge (as described above) occurs.

The output is always sample-based.

Dialog Box



Trigger count

The number of samples for which the block outputs the active value. Tunable.

Active level

The value to output after the first N sample times, `0` or `1`. Tunable.

Reset input

Enables the `Rst` input port.

Trigger type

The type of event that triggers a reset when the `Rst` port is enabled. Tunable.

Sample time

The sample period, T_s, for the block's counter. The block switches from the active value to the inactive value at $t=T_s*(N+1)$.

See Also

Counter	DSP Blockset
N-Sample Switch	DSP Blockset

See "Creating Signals Using Signal Generator Blocks" on page 3-36 for related information.

N-Sample Switch

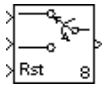
Purpose

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 (i.e., the output sample period is not inherited from the sample period of either input). The block applies a zero-order hold at the input ports, so the value the block reads from a given port between input sample times is the value of the most recent input to that port.

Both inputs must have the same dimension, except in the following two cases:

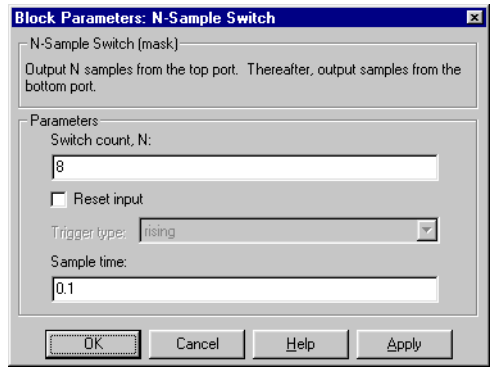
- When one input is a scalar, the block expands the scalar input to match the size of the other input.
- When one input is a 1-D vector and the other input is a row or column vector with the same number of elements, the block reshapes the 1-D vector to match the dimension of the other input.

The inputs must either both be frame-based or both be sample-based.

The **Reset input** check box enables the `Rst` input port. At any time during the count, a trigger event at the `Rst` port resets the counter to zero. The triggering event is specified by the **Trigger type** pop-up menu, and can be one of the following:

- **Rising edge** triggers the reset 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 the reset 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 the reset when either a rising or falling edge (as described above) occurs.

Dialog Box



Switch count

The number of sample periods, N, for which the output is connected to the top input before switching to the bottom input.

Reset input

Enables the `Rst` input port when selected.

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 $t=T_s*(N+1)$.

See Also

Counter	DSP Blockset
N-Sample Enable	DSP Blockset

Normalization

Purpose

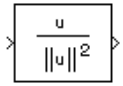
Normalize an input by its 2-norm or squared 2-norm.

Library

Math Functions / Math Operations

Description

The Normalization block independently normalizes each column of the M-by-N matrix input, *u*.



2-Norm

When the **Norm** parameter specifies **2-norm**, the block normalizes the *j*th 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 *j*th input column.

$$||u||_j = \sqrt{|u_1|^2 + |u_2|^2 + \cdots + |u_M|^2}$$

Equivalently,

```
y = u ./ (norm(u) + b)       % Equivalent MATLAB code
```

The normalization bias, *b*, is typically chosen to be a small positive constant (e.g., 1e-10) that prevents potential division by zero.

Squared 2-Norm

When the **Norm** parameter specifies **Squared 2-norm**, the block normalizes the *j*th input column as follows.

$$y_{ij} = \frac{u_{ij}}{||u||_j^2 + b}$$

where

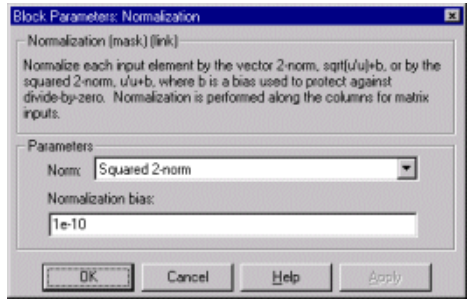
$$||u||_j^2 = |u_1|^2 + |u_2|^2 + \cdots + |u_M|^2$$

Equivalently,

```
y = u ./ (norm(u).^2 + b)    % Equivalent MATLAB code
```

The output has the same dimension and frame status as the input. 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, and the output retains the dimensions of the input.

Dialog Box



Norm

The type of normalization to apply, **2-norm** or **Squared 2-norm**. Tunable, except in Simulink's external mode.

Normalization bias

The value *b* to be added in the denominator to avoid division by zero. Tunable, except in Simulink's external mode.

See Also

Matrix Scaling	DSP Blockset
Reciprocal Condition	DSP Blockset
`norm`	MATLAB

Overlap-Add FFT Filter

Purpose

Implement the overlap-add method of frequency-domain filtering.

Library

Filtering / Filter Structures

Description

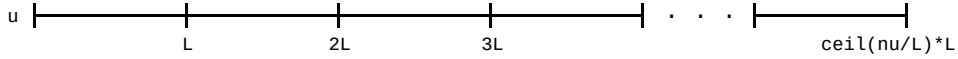
The Overlap-Add FFT Filter block uses an FFT to implement the *overlap-add method*, a technique that combines successive frequency-domain filtered sections of an input sequence.

Valid inputs to this block are 1-D vectors, sample-based vectors, frame-based vectors, and frame-based full matrices. All outputs are unbuffered into sample-based row vectors. The length of the output vector is equal to the number of channels in the input vector. An M-by-1 *sample-based* input has M channels, so it would result in a length-M sample-based output vector. An M-by-1 *frame-based* input has only one channel, so would result in a 1-by-1 (scalar) output.

The block's data output rate is M times faster than its data input rate, where M is the input frame-size. Thus, the block's data input and output rates are the same when the inputs are 1-D vectors, sample-based vectors, or frame-based row vectors. For frame-based column and frame-based full-matrix inputs, the block's data output rate is M times greater than the block's data input rate.

1-D vectors are treated as length-N sample-based vectors, and result in sample-based length-N row vectors.

The block breaks the scalar input sequence `u`, of length `nu`, into length-`L` nonoverlapping data sections,



which it linearly convolves with the filter's FIR coefficients,

$$H(z) = B(z) = b_1 + b_2 z^{-1} + \ldots + b_{n+1} z^{-n}$$

The numerator coefficients for H(*z*) are specified as a vector by the **FIR coefficients** parameter. The coefficient vector, `b = [b(1) b(2) ... b(n+1)]`, can be generated by one of the filter design functions in the Signal Processing Toolbox, such as `fir1`. All filter states are internally initialized to zero.

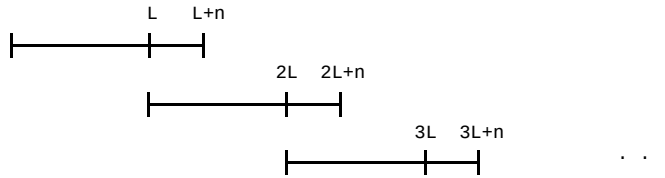
If either the filter coefficients or the inputs to the block are complex, the **Output** parameter should be set to **Complex**. Otherwise, the default **Output** setting, **Real**, instructs the block to take only the real part of the solution.

The block's overlap-add operation is equivalent to

```
y = ifft(fft(u(i:i+L-1),nfft) .* fft(b,nfft))
```

where `nfft` is specified by the **FFT size** parameter as a power-of-two value greater (typically *much* greater) than `n+1`. Values for **FFT size** that are not powers of two are rounded upwards to the nearest power-of-two value to obtain `nfft`.

The block overlaps successive output sections by `n` points and sums them.



The first `L` samples of each summation are output in sequence. The block chooses the parameter `L` based on the filter order and the FFT size.

```
L = nfft - n
```

Latency

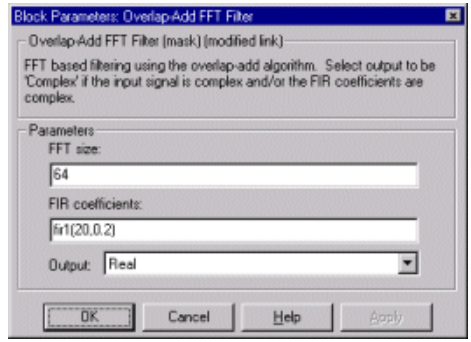
In *single-tasking* operation, the Overlap-Add FFT Filter block has a latency of `nfft-n+1` samples. The first `nfft-n+1` consecutive outputs from the block are zero; the first filtered input value appears at the output as sample `nfft-n+2`.

In *multitasking* operation, the Overlap-Add FFT Filter block has a latency of `2*(nfft-n+1)` samples. The first `2*(nfft-n+1)` consecutive outputs from the block are zero; the first filtered input value appears at the output as sample `2*(nfft-n)+3`.

See "Excess Algorithmic Delay (Tasking Latency)" on page 3-91 and "The Simulation Parameters Dialog Box" in the Simulink documentation for more information about block rates and Simulink's tasking modes.

Overlap-Add FFT Filter

Dialog Box



FFT size

The size of the FFT, which should be a power-of-two value greater than the length of the specified FIR filter.

FIR coefficients

The filter numerator coefficients.

Output

The complexity of the output; **Real** or **Complex**. If the input signal or the filter coefficients are complex, this should be set to **Complex.**

References

Oppenheim, A. V. and R. W. Schafer. *Discrete-Time Signal Processing.* Englewood Cliffs, NJ: Prentice Hall, 1989.

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

See Also

Direct-Form II Transpose Filter	DSP Blockset
Overlap-Save FFT Filter	DSP Blockset

See “Filter Structures” on page 4-23 for related information.

Purpose

Implement the overlap-save method of frequency-domain filtering.

Library

Filtering / Filter Structures

Description

The Overlap-Save FFT Filter block uses an FFT to implement the *overlap-save method*, a technique that combines successive frequency-domain filtered sections of an input sequence.

Valid inputs to this block are 1-D vectors, sample-based vectors, frame-based vectors, and frame-based full matrices. All outputs are unbuffered into sample-based row vectors. The length of the output vector is equal to the number of channels in the input vector. An M-by-1 sample-based input has M channels, so it would result in a length-M sample-based output vector. An M-by-1 frame-based input has only one channel, so would result in a 1-by-1 (scalar) output.

The block's data output rate is M times faster than its data input rate, where M is the input frame-size. Thus, the block's data input and output rates are the same when the inputs are 1-D vectors, sample-based vectors, or frame-based row vectors. For frame-based column and frame-based full-matrix inputs, the block's data output rate is M times greater than the block's data input rate.

1-D vectors are treated as length-N sample-based vectors, and result in sample-based length-N row vectors.

Overlapping sections of input `u` are circularly convolved with the FIR filter coefficients

$$H(z) = B(z) = b_1 + b_2 z^{-1} + \ldots + b_{n+1} z^{-n}$$

The numerator coefficients for $H(z)$ are specified as a vector by the **FIR coefficients** parameter. The coefficient vector, `b = [b(1) b(2) ... b(n+1)]`, can be generated by one of the filter design functions in the Signal Processing Toolbox, such as `fir1`. All filter states are internally initialized to zero.

If either the filter coefficients or the inputs to the block are complex, the **Output** parameter should be set to **Complex**. Otherwise, the default **Output** setting, **Real**, instructs the block to take only the real part of the solution.

The circular convolution of each section is computed by multiplying the FFTs of the input section and filter coefficients, and computing the inverse FFT of the product.

```
y = ifft(fft(u(i:i+(L-1)),nfft) .* fft(b,nfft))
```

where `nfft` is specified by the **FFT size** parameter as a power-of-two value greater (typically *much* greater) than `n+1`. Values for **FFT size** that are not powers of two are rounded upwards to the nearest power-of-two value to obtain `nfft`.

The first `n` points of the circular convolution are invalid and are discarded. The Overlap-Save FFT Filter block outputs the remaining `nfft-n` points, which are equivalent to the linear convolution.

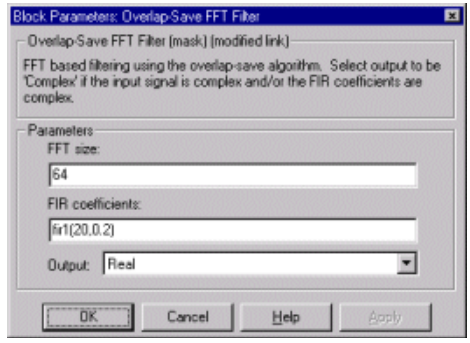
Latency

In *single-tasking* operation, the Overlap-Save FFT Filter block has a latency of `nfft-n+1` samples. The first `nfft-n+1` consecutive outputs from the block are zero; the first filtered input value appears at the output as sample `nfft-n+2`.

In *multitasking* operation, the Overlap-Save FFT Filter block has a latency of `2*(nfft-n+1)` samples. The first `2*(nfft-n+1)` consecutive outputs from the block are zero; the first filtered input value appears at the output as sample `2*(nfft-n)+3`.

See "Excess Algorithmic Delay (Tasking Latency)" on page 3-91 and "The Simulation Parameters Dialog Box" in the Simulink documentation for more information about block rates and Simulink's tasking modes.

Dialog Box



FFT size

The size of the FFT, which should be a power-of-two value greater than the length of the specified FIR filter.

FIR coefficients

The filter numerator coefficients.

Output

The complexity of the output; **Real** or **Complex**. If the input signal or the filter coefficients are complex, this should be set to **Complex.**

References

Oppenheim, A. V. and R. W. Schafer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989.

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

See Also

Direct-Form II Transpose Filter	DSP Blockset
Overlap-Add FFT Filter	DSP Blockset

See “Filter Structures” on page 4-23 for related information.

Pad

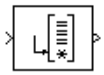
Purpose

Alter the input size by padding or truncating rows and/or columns.

Library

Signal Operations

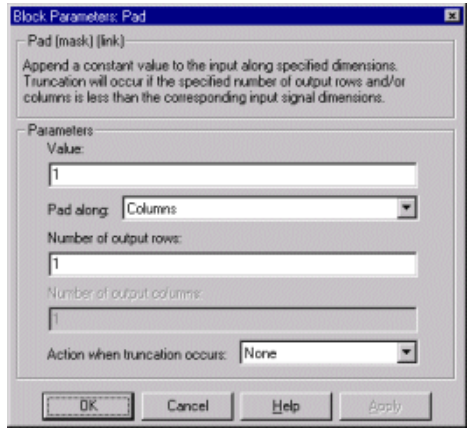
Description



The Pad block changes the size of the input matrix from M_i-by-N_i to M_o-by-N_o by padding or truncating along the rows, the columns, or both dimensions. The dimensions of the output, M_o and N_o, are specified by the **Number of output rows** and **Number of output columns** parameters, respectively. The value with which to pad the input is set by the **Value** parameter.

The behavior of the Pad block and Zero Pad block are identical, with the exception that the Pad block can pad the input matrix with values other than zero. See the Zero Pad block reference for more information on the behavior of the Pad block.

Dialog Box



Value

The scalar value with which to pad the input matrix.

Pad along

The direction along which to pad or truncate. **Columns** specifies that the *row* dimension should be changed to M_o. **Rows** specifies that the *column* dimension should be changed to N_o. **Columns and rows** specifies that both column and row dimensions should be changed. **None** disables padding and truncation and passes the input through to the output unchanged.

Number of output rows

The desired number of rows in the output, M_o. This parameter is enabled when **Columns** or **Columns and rows** is selected in the **Pad along** menu.

Number of output columns

The desired number of columns in the output, N_o. This parameter is enabled when **Rows** or **Columns and rows** is selected in the **Pad along** menu.

See Also

Matrix Concatenation	Simulink
Repeat	DSP Blockset
Submatrix	DSP Blockset
Upsample	DSP Blockset
Variable Selector	DSP Blockset
Zero Pad	DSP Blockset

Permute Matrix

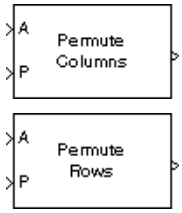
Purpose

Reorder the rows or columns of a matrix.

Library

Math Functions / Matrices and Linear Algebra / Matrix Operations

Description



The Permute Matrix block reorders the rows or columns of M-by-N input matrix `A` as specified by indexing input `P`.

When the **Permute** parameter is set to **Rows**, the block uses the rows of `A` to create a new matrix with the same column dimension. Input `P` is a length-L vector whose elements determine where each row from `A` should be placed in the L-by-N output matrix.

```
% Equivalent MATLAB code
y = [A(P(1),:) ; A(P(2),:) ; A(P(3),:) ; ... ; A(P(end),:)]
```

For row permutation, a length-M 1-D vector input at the `A` port is treated as a M-by-1 matrix.

When the **Permute** parameter is set to **Columns**, the block uses the columns of `A` to create a new matrix with the same row dimension. Input `P` is a length-L vector whose elements determine where each column from `A` should be placed in the M-by-L output matrix.

```
% Equivalent MATLAB code
y = [A(:,P(1)) A(:,P(2)) A(:,P(3)) ... A(:,P(end))]
```

For column permutation, a length-N 1-D vector input at the `A` port is treated as a 1-by-N matrix.

When an index value in input `P` references a nonexistent row or column of matrix `A`, the block reacts with the behavior specified by the **Invalid permutation index** parameter. The following options are available:

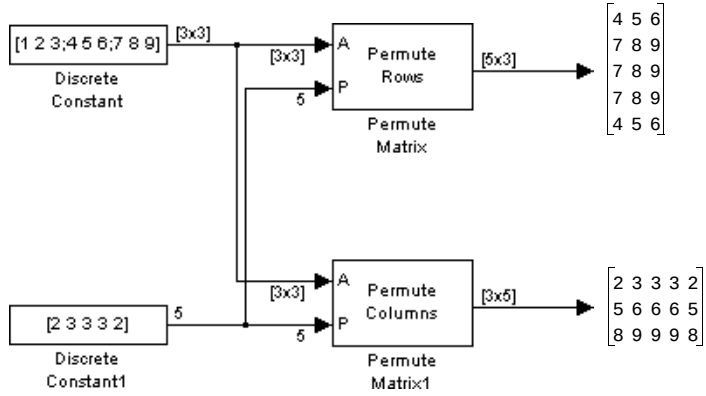
- **Clip index** – Clip the index to the nearest valid value (1 or M for row permutation, and 1 or N for column permutation), and *do not* issue an alert. Example: For a 3-by-7 input matrix, a column index of 9 is clipped to 7, and a row index of -2 is clipped to 1.
- **Clip and warn** – Display a warning message in the MATLAB command window, and clip the index as described above.
- **Generate error** – Display an error dialog box and terminate the simulation.

When length of the permutation vector P is not equal to the number of rows or columns of the input matrix A, you can choose to get an error dialog box and terminate the simulation by checking **Error when length of P is not equal to Permute dimension size**.

If input A is frame-based, the output is frame-based; otherwise, the output is sample-based.

Example

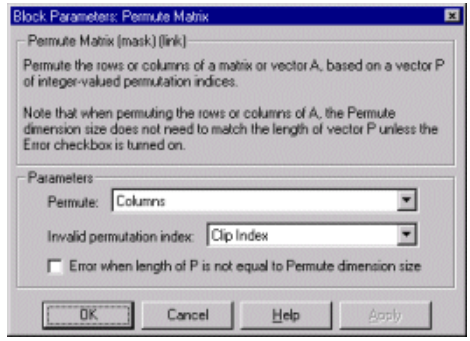
In the model below, the top Permute Matrix block places the second row of the input matrix in both the first and fifth rows of the output matrix, and places the third row of the input matrix in the three middle rows of the output matrix. The bottom Permute Matrix block places the second column of the input matrix in both the first and fifth columns of the output matrix, and places the third column of the input matrix in the three middle columns of the output matrix.



As shown in the example above, rows and columns of A can appear any number of times in the output, or not at all.

Permute Matrix

Dialog Box



Permute

Method of constructing the output matrix; by permuting rows or columns of the input.

Invalid permutation index

Response to an invalid index value. Tunable, except in Simulink's external mode.

Error when length of P is not equal to Permute dimension size

Option to display an error dialog box and terminate the simulation if the length of the permutation vector `P` is not equal to the number of rows or columns of the input matrix `A`.

See Also

Submatrix	DSP Blockset
Transpose	DSP Blockset
Variable Selector	DSP Blockset
`permute`	MATLAB

See "Reordering Channels in a Frame-Based Multichannel Signal" on page 3-61 for related information.

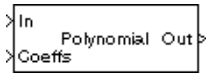
Purpose

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.

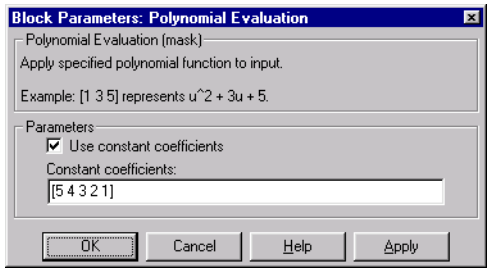
The table below shows some examples of the block's operation for various coefficient vectors.

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.

Polynomial Evaluation

Dialog Box



Use constant coefficients

When selected, enables the **Constant coefficients** parameter and disables the `Coeffs` input port.

Constant coefficients

The vector of polynomial coefficients to apply to the input, in order of descending exponents. This parameter is enabled when the **Use constant coefficients** check box is selected.

See Also

Least Squares Polynomial Fit	DSP Blockset
Math Function	Simulink
Sum	Simulink
`polyval`	MATLAB

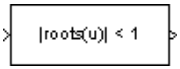
Purpose

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,

$$f(x) = u_1 x^{M-1} + u_2 x^{M-2} + \cdots + u_M$$

arranged in order of descending exponents, u_1, u_2, ..., u_M. The polynomial has order M-1 and positive integer exponents.

Inputs can be frame-based or sample-based, and both represent the polynomial coefficients as shown above. For convenience, a length-M 1-D vector input is treated as an M-by-1 matrix.

The output is a 1-by-N matrix with each column containing the value `1` or `0`. The value `1` indicates that the polynomial in the corresponding column of the input is stable; i.e., the magnitudes of all solutions to $f(x) = 0$ are less than 1. The value `0` indicates that the polynomial in the corresponding column of the input may be unstable; i.e., the magnitude of at least one solution to $f(x) = 0$ is greater than or equal to 1.

The output is always sample-based.

Applications

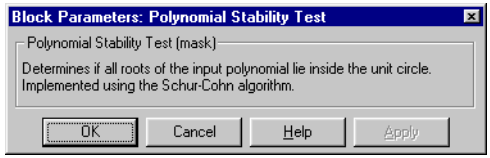
This block is most commonly used to check the pole locations of the denominator polynomial, $A(z)$, of a transfer function, $H(z)$.

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_1 + b_2 z^{-1} + \ldots + b_m z^{-(m-1)}}{a_1 + a_2 z^{-1} + \ldots + a_n z^{-(n-1)}}$$

The poles are the n-1 roots of the denominator polynomial, $A(z)$. If any poles are located outside the unit circle, the transfer function $H(z)$ is unstable. As is

typical in DSP applications, the transfer function above is specified in descending powers of z^{-1} rather than z.

Dialog Box



See Also

Least Squares Polynomial Fit	DSP Blockset
Polynomial Evaluation	DSP Blockset
`polyfit`	MATLAB

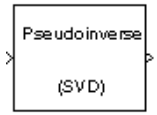
Purpose

Compute the Moore-Penrose pseudoinverse of a matrix.

Library

Math Functions / Matrices and Linear Algebra / Matrix Inverses

Description



The Pseudoinverse block computes the Moore-Penrose pseudoinverse of input matrix A.

```
[U,S,V] = svd(A,0)          % Equivalent MATLAB code
```

The pseudoinverse of A is the matrix A^+ such that

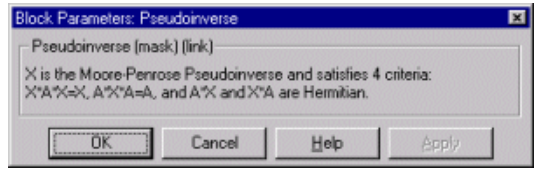
$$A^+ = VS^+U^*$$

where U and V are orthogonal matrices, and S is a diagonal matrix. The pseudoinverse has the following properties:

- $AA^+ = (AA^+)^*$
- $A^+A = (A^+A)^*$
- $AA^+A = A$
- $A^+AA^+ = A^+$

The output is always sample-based.

Dialog Box



References

Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

See Also

Cholesky Inverse	DSP Blockset
LDL Inverse	DSP Blockset
LU Inverse	DSP Blockset
Singular Value Decomposition	DSP Blockset
`inv`	MATLAB

Pseudoinverse

See “Inverting Matrices” on page 4-34 for related information.

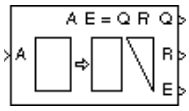
Purpose

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 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.

$$\left|r_{i+1,j+1}\right| > \left|r_{i,j}\right| \qquad i = j$$

QR factorization is an important tool for solving linear systems of equations because of good error propagation properties and the invertability of unitary matrices.

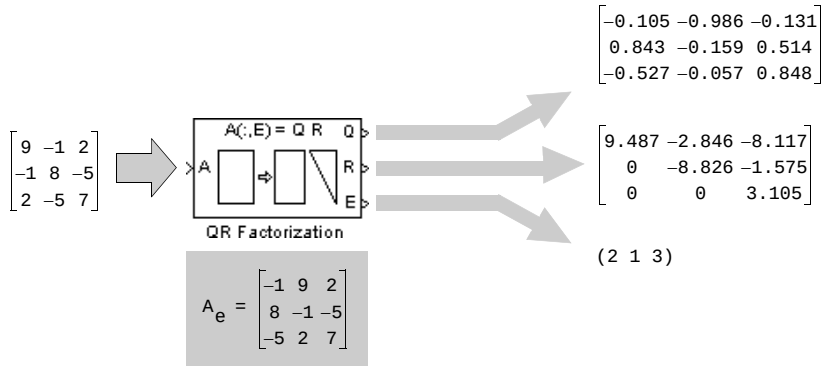
$$Q^{-1} = Q^{*}$$

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.

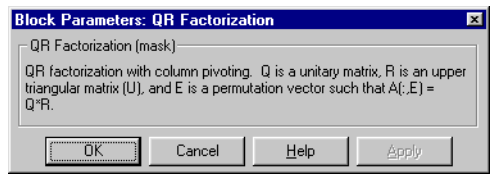
Example

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.

QR Factorization



Dialog Box



References

Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

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 4-32 for related information.

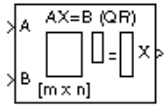
Purpose

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-hand-side M-by-L matrix, B. A length-M 1-D vector input at either port is treated as an M-by-1 matrix.

The output at the x port is the N-by-L matrix, *X*. *X* is always sample based, and is chosen to minimize the sum of the squares of the elements of B-A*X*. When B is a vector, this solution minimizes the vector 2-norm of the residual (B-A*X* 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 A*X*=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

$$A_eX = B_e,$$

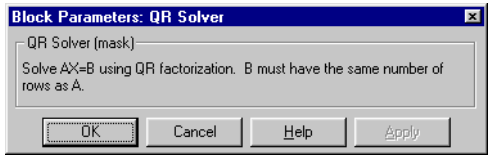
and

$$QRX = B_e$$

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



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 4-31 for related information.

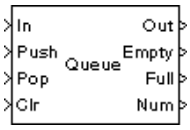
Purpose

Store inputs in a FIFO register.

Library

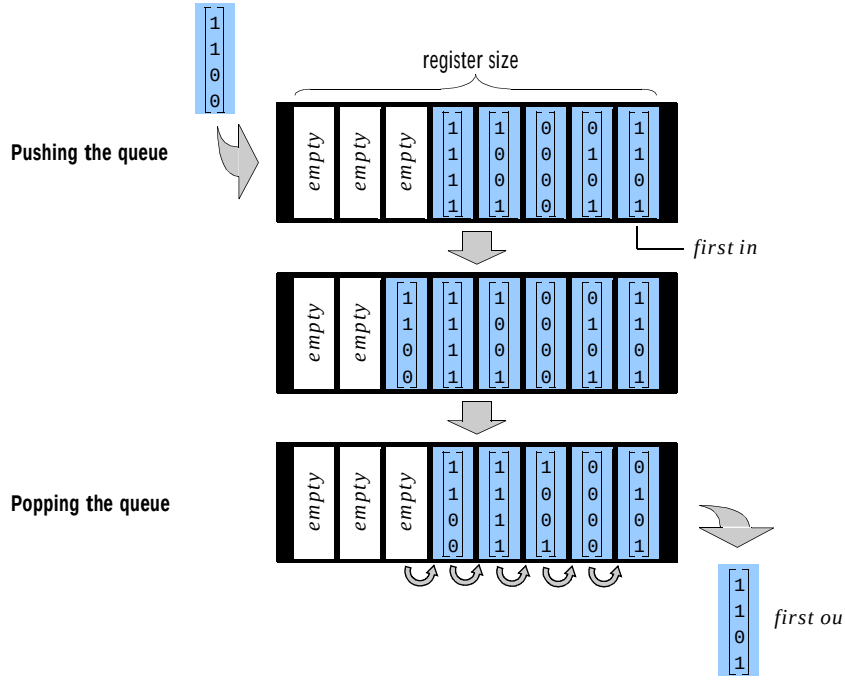
Signal Management / Buffers

Description



The Queue block stores a sequence of input samples in a FIFO (first in, first out) register. The register capacity is set by the **Register size** parameter, and inputs can be scalars, vectors, or matrices.

The block *pushes* the input at the `In` port onto the end of the queue when a trigger event is received at the `Push` port. When a trigger event is received at the `Pop` port, the block *pops* the first element off the queue and holds the `Out` port at that value. The first input to be pushed onto the queue is always the first to be popped off.



A trigger event at the optional `Clr` port (enabled by the **Clear input** check box) empties the queue contents. If **Clear output port on reset** is selected, then a trigger event at the `Clr` port empties the queue *and* sets the value at the `Out` port to zero. This setting also applies when a disabled subsystem containing

the Queue block is reenabled; the `Out` port value is only reset to zero in this case if **Clear output port on reset** is selected.

When two or more of the control input ports are triggered at the same time step, the operations are executed in the following order:

1. `Clr`
2. `Push`
3. `Pop`

The 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 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 **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.

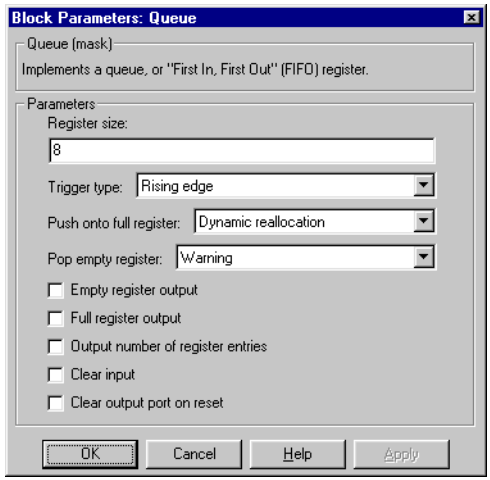
In	Push	Pop	Clr	Queue	Out	Empty	Full	Num
1	0	0	0	*top* [ ] *bottom*	0	1	0	0
2	1	0	0	*top* [2] *bottom*	0	0	0	1
3	0	0	0	*top* [3 2] *bottom*	0	0	0	2
4	1	0	0	*top* [4 3 2] *bottom*	0	0	0	3
5	0	0	0	*top* [5 4 3 2] *bottom*	0	0	1	4
6	0	1	0	*top* [5 4 3] *bottom*	2	0	0	3
7	0	0	0	*top* [5 4] *bottom*	3	0	0	2
8	0	1	0	*top* [5] *bottom*	4	0	0	1
9	0	0	0	*top* [ ] *bottom*	5	1	0	0
10	1	0	0	*top* [10] *bottom*	5	0	0	1
11	0	0	0	*top* [11 10] *bottom*	5	0	0	2
12	1	0	1	*top* [12] *bottom*	0	0	0	1

Note that at the last step shown, the `Push` and `Clr` ports are triggered simultaneously. The `Clr` trigger takes precedence, and the queue is first cleared and then pushed.

Example 2

The `dspqdemo` demo provides another example of Queue operation.

Dialog Box



Register size

The number of entries that the FIFO register can hold.

Trigger type

The type of event that triggers the block's execution.

Push onto full register

Response to a trigger received at the `Push` port when the register is full.

Pop empty register

Response to a trigger received at the `Pop` port when the register is empty. 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.

Clear input

Enable the `Clr` input port, which empties the queue when the trigger specified by the **Trigger type** is received.

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.

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.

Distribution Type

When the **Source type** parameter is set to **Uniform**, the output samples are drawn from a uniform distribution whose minimum and maximum values are specified by the **Minimum** and **Maximum** parameters, respectively. All values in this range are equally likely to be selected. A length-N vector specified for one or both of these parameters generates an N-channel output (M-by-N matrix) containing a unique random distribution in each channel.

For example, specify

- **Minimum** = `[ 0  0 -3 -3]`
- **Maximum** = `[10 10 20 20]`

to generate a four-channel output whose first and second columns contain random values in the range [0, 10], and whose third and fourth columns contain random values in the range [-3, 20]. When only one of the **Minimum** and **Maximum** parameters is specified as a vector, the other is scalar expanded to the same length.

When the **Source type** parameter is set to **Gaussian**, the output samples are drawn from the normal distribution defined by the **Mean** and **Variance** parameters. A length-N vector specified for one or both of the **Mean** and **Variance** parameters generates an N-channel output (M-by-N frame matrix) containing a distinct random distribution in each column. When only one of these parameters is specified as a vector, the other is scalar expanded to the same length.

Output Data Type

The block's output can be either real or complex, as selected by the **Real** and **Complex** options in the **Output complexity** parameter. (These settings control all channels of the output, so real and complex data cannot be combined

in the same output.) For complex output with a **Uniform** distribution, the real and imaginary components in each channel are both drawn from the same uniform random distribution, defined by the **Minimum** and **Maximum** parameters for that channel.

For complex output with a **Gaussian** distribution, the real and imaginary components in each channel are drawn from normal distributions with different means. In this case, the **Mean** parameter for each channel should specify a complex value; the real component of the **Mean** parameter specifies the mean of the real components in the channel, while the imaginary component specifies the mean of the imaginary components in the channel. If either the real or imaginary component is omitted from the **Mean** parameter, a default value of 0 is used for the mean of that component.

For example, a **Mean** parameter setting of `[5+2i 0.5 3i]` generates a three-channel output with the following means.

Channel 1 mean	*real* = 5	*imaginary* = 2
Channel 2 mean	*real* = 0.5	*imaginary* = 0
Channel 3 mean	*real* = 0	*imaginary* = 3

For complex output, the **Variance** parameter, σ^2, specifies the *total variance* for each output channel. This is the sum of the variances of the real and imaginary components in that channel.

$$\sigma^2 = \sigma_{Re}^2 + \sigma_{Im}^2$$

The specified variance is equally divided between the real and imaginary components, so that

$$\sigma_{Re}^2 = \frac{\sigma^2}{2}$$

$$\sigma_{Im}^2 = \frac{\sigma^2}{2}$$

Initial Seed

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.

For real outputs (**Output complexity** parameter set to **Real**), a length-N seed vector can be specified to set a distinct initial generator seed for each individual channel. When a scalar seed is specified for a multichannel output, the block uses the specified seed for the first channel, and increments the seed by 2 for each additional channel. For example, specifying an **Initial seed** value of `10` for a five-channel output is equivalent to specifying an **Initial seed** vector of `[10 12 14 16 18]`.

For complex outputs (**Output complexity** parameter set to **Complex**), a length-N seed vector can be specified to set a distinct initial generator seed to be used for the *real components* of each individual channel. The block increments these values by 1 to determine the initial seeds used for the imaginary components of the corresponding channels.

When a scalar seed is specified for a multichannel complex output, the block uses the specified seed for the *real components* of the first channel, and increments the seed by 2 for the *real components* of each additional channel. The block increments the specified seed by 1 for the imaginary components of the first channel, and increments the first channel's imaginary seed by 2 for the imaginary components of each additional channel. For example, specifying an **Initial seed** value of `10` for a five-channel complex output is equivalent to specifying an **Initial seed** vector of `[10 12 14 16 18]`. These values are used to seed the real-component generator for each channel; the vector `[11 13 15 17 19]` is used to seed the imaginary-component generator for each channel.

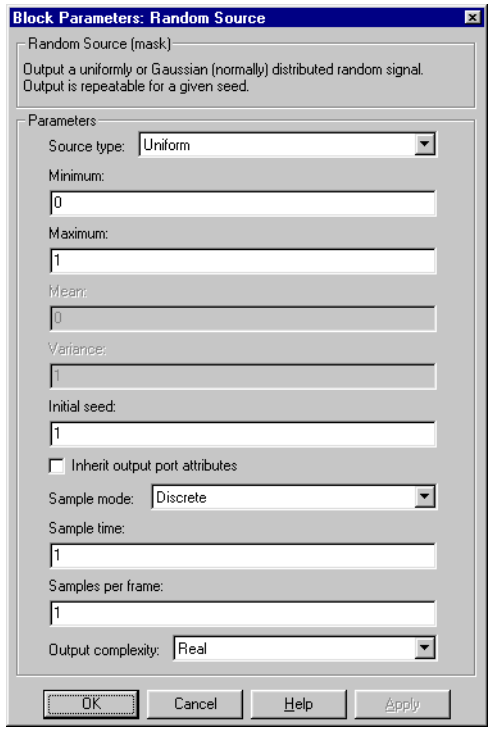
Sample Period

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



Source type

The distribution from which to draw the random values, **Uniform** or **Gaussian**.

Minimum

The minimum value in the uniform distribution. This parameter is only enabled when **Uniform** is selected from the **Source type** parameter. Tunable.

Maximum

The maximum value in the uniform distribution. This parameter is only enabled when **Uniform** is selected from the **Source type** parameter. Tunable.

Mean

The mean of the Gaussian (normal) distribution. This parameter is only enabled when **Gaussian** is selected from the **Source type** parameter. Tunable.

Variance

The variance of the Gaussian (normal) distribution. This parameter is only enabled when **Gaussian** is selected from the **Source type** parameter. Tunable.

Initial seed

The initial seed(s) to use for the random number generator.

Inherit output port attributes

When selected, allows the block to inherit the sample mode, sample period, and complexity of a downstream block. (The **Sample mode**, **Sample time**, **Samples per frame**, and **Output complexity** parameters are disabled.) The output is a length-M sample-based 1-D vector, where length M is inherited from the downstream block. If the **Minimum**, **Maximum**, **Mean**, or **Variance** parameter specifies N channels, the 1-D vector output contains M/N samples from each channel. An error occurs in this case if M is not an integer multiple of N.

Sample mode

The sample mode, **Continuous** or **Discrete**. This parameter is enabled when the **Inherit output port attributes** check box is deselected.

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 deselected.

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 deselected.

Output complexity

The data type of the output, **Real** or **Complex**. This parameter is enabled when the **Inherit output port attributes** check box is deselected.

See Also

Discrete Impulse	DSP Blockset
DSP Constant	DSP Blockset
Maximum	DSP Blockset
Minimum	DSP Blockset
Random Number	Simulink
Signal From Workspace	DSP Blockset
Signal Generator	Simulink
Standard Deviation	DSP Blockset
Variance	DSP Blockset
`rand`	MATLAB
`randn`	MATLAB

See “Creating Signals Using Signal Generator Blocks” on page 3-36 for related information.

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 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 output is a real M_o-by-N matrix, where M_o is specified by the **FFT length** parameter. Each output column contains the length-M_o real cepstrum of the corresponding input column.

```
y = real(ifft(log(abs(fft(u,Mo)))))    % Equivalent MATLAB code
```

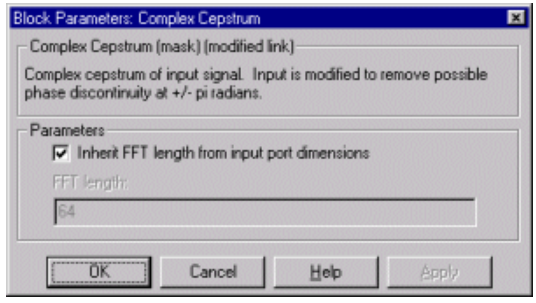
or, more compactly,

```
y = rceps(u,Mo)
```

When the **Inherit FFT length from input port dimensions** check box is selected, the output frame size matches the input frame size (M_o=M). In this case, a *sample-based* length-M row vector input is processed as a single channel (i.e., as an M-by-1 column vector), and the output is a length-M row vector. A 1-D vector input is *always* processed as a single channel, and the output is a 1-D vector.

The output is always sample-based, and the output port rate is the same as the input port rate.

Dialog Box



Inherit FFT length from input port dimensions

When selected, matches the output frame size to the input frame size.

FFT length

The number of frequency points at which to compute the FFT, which is also the output frame size, M_o. This parameter is available when **Inherit FFT length from input port dimensions** is not selected.

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

$$y = \frac{1}{\kappa} = \frac{1}{\left\|A^{-1}\right\|_1 \left\|A\right\|_1}$$

where κ is the condition number ($\kappa \geq 1$), and y is the scalar sample-based output ($0 \leq y < 1$).

The matrix 1-norm, $\left\|A\right\|_1$, is the maximum column-sum in the M-by-M matrix A.

$$\left\|A\right\|_1 = \max_{1 \leq j \leq M} \sum_{i=1}^{M} \left|a_{ij}\right|$$

For a 3-by-3 matrix:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

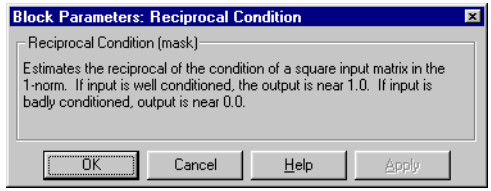
$$\left\|A\right\|_1 = \max(A_1, A_2, A_3)$$

$$\left|a_{13}\right| + \left|a_{23}\right| + \left|a_{33}\right| = A_3$$

$$\left|a_{12}\right| + \left|a_{22}\right| + \left|a_{32}\right| = A_2$$

$$\left|a_{11}\right| + \left|a_{21}\right| + \left|a_{31}\right| = A_1$$

Dialog Box



References

Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

See Also

Matrix 1-Norm	DSP Blockset
Normalization	DSP Blockset
`rcond`	MATLAB

Remez FIR Filter Design

Purpose

Design and apply an equiripple FIR filter.

Library

Filtering / Filter Designs

Description

The Remez FIR Filter Design block implements the Parks-McClellan algorithm to design and apply a linear-phase filter with an arbitrary multiband magnitude response. The filter design, which uses the `remez` function in the Signal Processing Toolbox, minimizes the maximum error between the desired frequency response and the actual frequency response. Such filters are called *equiripple* due to the equiripple behavior of their approximation error. The block applies the filter to a discrete-time input using the Direct-Form II Transpose Filter block.

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

The **Filter type** parameter allows you to specify one of the following filters:

- **Multiband**

  The multiband filter has an arbitrary magnitude response and linear phase.

- **Differentiator**

  The differentiator filter approximates the ideal differentiator. Differentiators are antisymmetric FIR filters with approximately linear magnitude responses. To obtain the correct derivative, scale the **Gains at these frequencies** vector by πF_s rad/s, where F_s is the sample frequency in Hertz.

- **Hilbert Transformer**

  The Hilbert transformer filter approximates the ideal Hilbert transformer. Hilbert transformers are antisymmetric FIR filters with approximately constant magnitude.

The **Band-edge frequency vector** parameter is a vector of frequency points in the range `0` to `1`, where `1` corresponds to half the sample frequency. Each band is defined by the two bounding frequencies, so this vector must have even length. Frequency points must appear in ascending order. The **Gains at these frequencies** parameter is a vector of the same size containing the desired

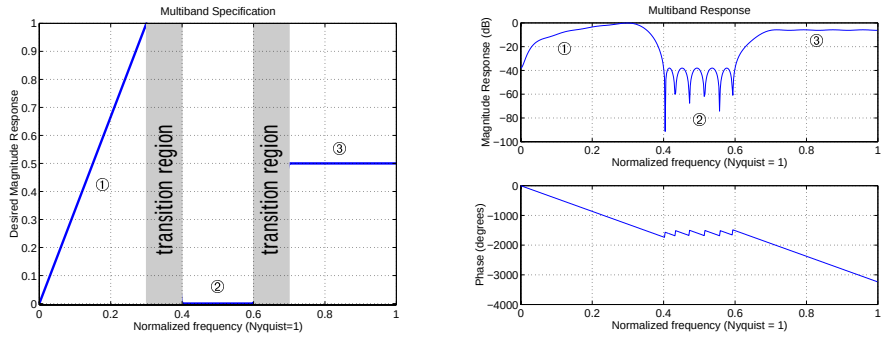
magnitude response at the corresponding points in the **Band-edge frequency vector**.

Each odd-indexed frequency-amplitude pair defines the left endpoint of a line segment representing the desired magnitude response in that frequency band. The corresponding even-indexed frequency-amplitude pair defines the right endpoint. Between the frequency bands specified by these end-points, there may be undefined sections of the specified frequency response. These are called "don't care" or "transition" regions, and the magnitude response in these areas is a by-product of the optimization in the other (specified) frequency ranges.

Band edge frequency = [0 0.3 0.4 0.6 0.7 1]

Gains = [0 1 0 0 0.5 0.5]

Band: ① ② ③



The **Weights** parameter is a vector that specifies the emphasis to be placed on minimizing the error in certain frequency bands relative to others. This vector specifies one weight per band, so it is half the length of the **Band-edge frequency vector** and **Gains at these frequencies** vectors.

In most cases, differentiators and Hilbert transformers have only a single band, so the weight is a scalar value that does not affect the final filter. However, the **Weights** parameter is useful when using the block to design an antisymmetric multiband filter, such as a Hilbert transformer with stopbands.

Examples

Example 1: Multiband

Consider a lowpass filter with a transition band in the normalized frequency range 0.4 to 0.5, and 10 times greater error minimization in the stopband than in the passband.

In this case:

- **Filter type** = **Multiband**
- **Band-edge frequency vector** = `[0 0.4 0.5 1]`
- **Gains at these frequencies** = `[1 1 0 0]`
- **Weights** = `[1 10]`

Example 2: Differentiator

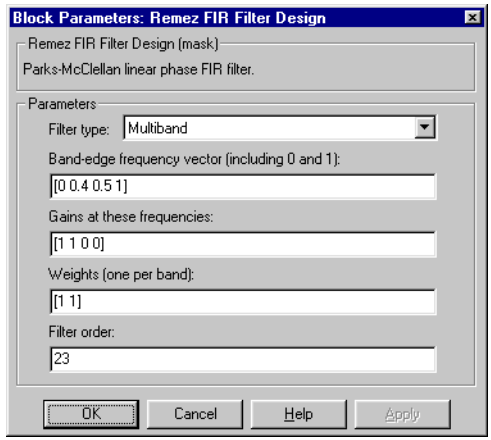
Assume the specifications for a differentiator filter require it to have order 21. The “ramp” response extends over the entire frequency range. In this case, specify:

- **Filter type** = **Differentiator**
- **Band-edge frequency vector** = `[0 1]`
- **Gains at these frequencies** = `[0 pi*Fs]`
- **Filter order** = `21`

For a type III (even order) filter, the differentiation band should stop short of half the sample frequency. For example, if the filter order is 20, you could specify the block parameters as follows:

- **Filter type** = **Differentiator**
- **Band-edge frequency vector** = `[0 0.9]`
- **Gains at these frequencies** = `[0 0.9*pi*Fs]`
- **Filter order** = `20`

Dialog Box



Filter type

The filter type. Tunable.

Band-edge frequency vector

A vector of frequency points, in ascending order, in the range `0` to `1`. The value `1` corresponds to half the sample frequency. This vector must have even length. Tunable.

Gains at these frequencies

A vector of frequency-response magnitudes corresponding to the points in the **Band-edge frequency** vector. This vector must be the same length as the **Band-edge frequency** vector. Tunable.

Weights

A vector containing one weight for each frequency band. This vector must be half the length of the **Band-edge frequency** and **Gains at these frequencies** vectors. Tunable.

Filter order

The filter order.

References

Oppenheim, A. V. and R. W. Schafer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989.

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

Remez FIR Filter Design

See Also

Digital FIR Filter Design	DSP Blockset
Least Squares FIR Filter Design	DSP Blockset
Yule-Walker IIR Filter Design	DSP Blockset
`firls`	Signal Processing Toolbox
`remez`	Signal Processing Toolbox

See “Filter Designs” on page 4-3 for related information.

Purpose

Resample an input at a higher rate by repeating values.

Library

Signal Operations

Description

The Repeat block upsamples each channel of the M_i-by-N input to a rate L times higher than the input sample rate by repeating each consecutive input sample L times at the output. The integer L is specified by the **Repetition count** parameter.

Sample-Based Operation

When the input is sample-based, the block treats each of the M∗N matrix elements as an independent channel, and upsamples each channel over time. The **Frame-based mode** parameter must be set to **Maintain input frame size**. The output sample rate is L times higher than the input sample rate ($T_{so} = T_{si}/L$), and the input and output sizes are identical.

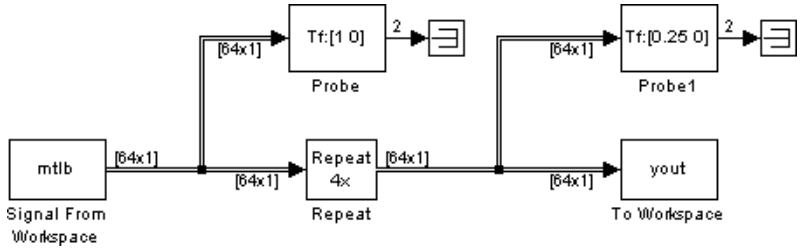
Frame-Based Operation

When the input is frame-based, the block treats each of the N input columns as a frame containing M_i sequential time samples from an independent channel. The block upsamples each channel independently by repeating each row of the input matrix L times at the output. The **Frame-based mode** parameter determines how the block adjusts the rate at the output to accommodate the repeated rows. There are two available options:

- **Maintain input frame size**

  The block generates the output at the faster (upsampled) rate by using a proportionally shorter frame *period* at the output port than at the input port. For L repetitions of the input, the output frame period is L times shorter than the input frame period ($T_{fo} = T_{fi}/L$), but the input and output frame sizes are equal.

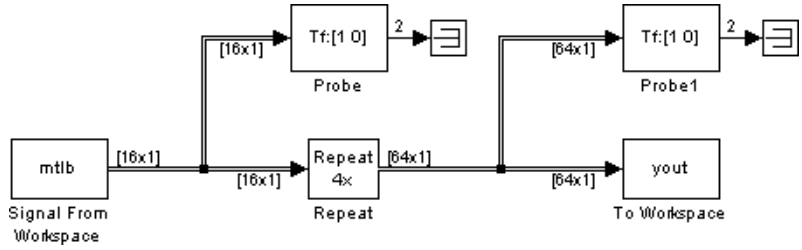
  The model below shows a single-channel input with a frame period of 1 second being upsampled through 4-times repetition to a frame period of 0.25 seconds. The input and output frame sizes are identical.



- **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 Simulink's 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.

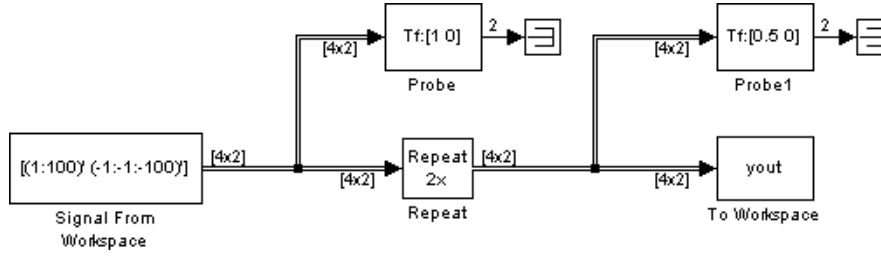
Nonzero Latency. The Repeat block has tasking latency only for multirate operation in Simulink's multitasking mode:

- In sample-based mode, the initial condition for each channel is repeated for the first L output samples. The channel's first input appears as output sample L+1. The **Initial condition** value can be an M_i-by-N matrix containing one value for each channel, or a scalar to be applied to all signal channels.
- In frame-based mode, the first row of the initial condition matrix is repeated for the first L output samples, the second row of the initial condition matrix is repeated for the next L output samples, and so on. The first row of the first input matrix appears in the output as sample M_iL+1. The **Initial condition** value can be an M_i-by-N matrix, or a scalar to be repeated across all elements of the M_i-by-N matrix. See the example below for an illustration of this case.

See "Excess Algorithmic Delay (Tasking Latency)" on page 3-91 and "The Simulation Parameters Dialog Box" in the Simulink documentation for more information about block rates and Simulink's 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 deselecting 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 Simulink's multitasking mode, select **Fixed-step** and **discrete** from the **Type** controls in the **Solver** panel of the **Simulation Parameters** dialog box, and select **MultiTasking** from the **Mode** parameter. Also set the **Stop time** to `30`.

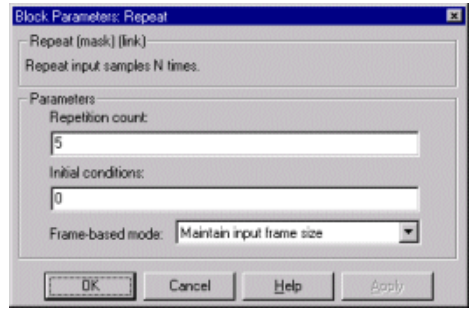
Run the model and look at the output, `yout`. The first few samples of each channel are shown below.

```
yout =

    11   -11
    11   -11
    12   -12
    12   -12
    13   -13
    13   -13
    14   -14
```

```
14   -14
 1    -1
 1    -1
 2    -2
 2    -2
 3    -3
 3    -3
 4    -4
 4    -4
 5    -5
 5    -5
```

Since we ran this frame-based multirate model in multitasking mode, the block repeats each row of the initial condition matrix for L output samples, where L is the **Repetition count** of 2. The first row of the first input matrix appears in the output as sample 9 (i.e., sample M_iL+1, where M_i is the input frame size).

Dialog Box



Repetition count

The integer number of times, L, that the input value is repeated at the output. This is the factor by which the output frame size or sample rate is increased.

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** (i.e., increase the frame rate), or **Maintain input frame rate** (i.e., increase the frame size). The **Frame-based mode** parameter must be set to **Maintain input frame size** for sample-base inputs.

See Also

FIR Interpolation	DSP Blockset
Upsample	DSP Blockset
Zero Pad	DSP Blockset

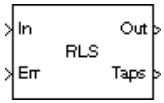
Purpose

Compute filter estimates for an input using the RLS adaptive filter algorithm.

Library

Filtering / Adaptive Filters

Description



The RLS Adaptive Filter block recursively computes the least squares estimate (RLS) of the FIR filter coefficients.

The corresponding RLS filter is expressed in matrix form as

$$k(n) = \frac{\lambda^{-1}P(n-1)u(n)}{1+\lambda^{-1}u^{H}(n)P(n-1)u(n)}$$

$$y(n) = \hat{w}^{H}(n-1)u(n)$$

$$e(n) = d(n)-y(n)$$

$$\hat{w}(n) = \hat{w}(n-1)+k(n)e^{*}(n)$$

$$P(n) = \lambda^{-1}P(n-1)-\lambda^{-1}k(n)u^{H}(n)P(n-1)$$

where λ^{-1} denotes the reciprocal of the exponential weighting factor. The variables are as follows.

Variable	Description
n	The current algorithm iteration
$u(n)$	The buffered input samples at step n
$P(n)$	The inverse correlation matrix at step n
$k(n)$	The gain vector at step n
$\hat{w}(n)$	The vector of filter-tap estimates at step n
$y(n)$	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

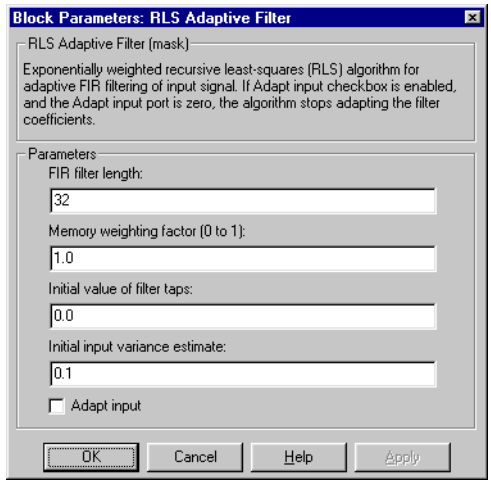
$$I\frac{1}{\hat{\sigma}^2}$$

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



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.

See Also

Kalman Adaptive Filter	DSP Blockset
LMS Adaptive Filter	DSP Blockset

See “Adaptive Filters” on page 4-3 for related information.

RMS

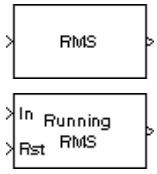
Purpose

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.^2)/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.

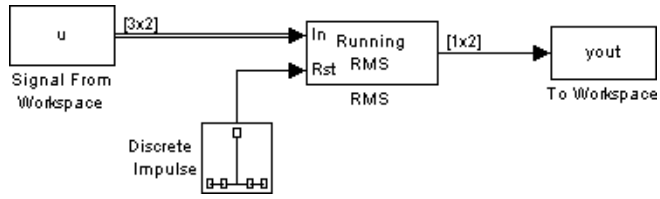
If the **Reset port** parameter is set to **Non-zero sample**, the optional `Rst` port is enabled and the block resets the running RMS when the scalar input at the `Rst` port is nonzero. (The `Rst` port can be disabled by setting the **Reset port**

parameter to **None**.) 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.

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.

Example

The RMS block in the model below calculates the running RMS of a frame-based 3-by-2 (two-channel) matrix input, `u`. The running RMS is reset at *t*=2 by an impulse to the block's `Rst` port.



The RMS block has the following settings:

- **Running RMS** = ☑
- **Reset port** = **Non-zero sample**

The Signal From Workspace block has the following settings:

- **Signal** = `u`
- **Sample time** = `1/3`
- **Samples per frame** = `3`

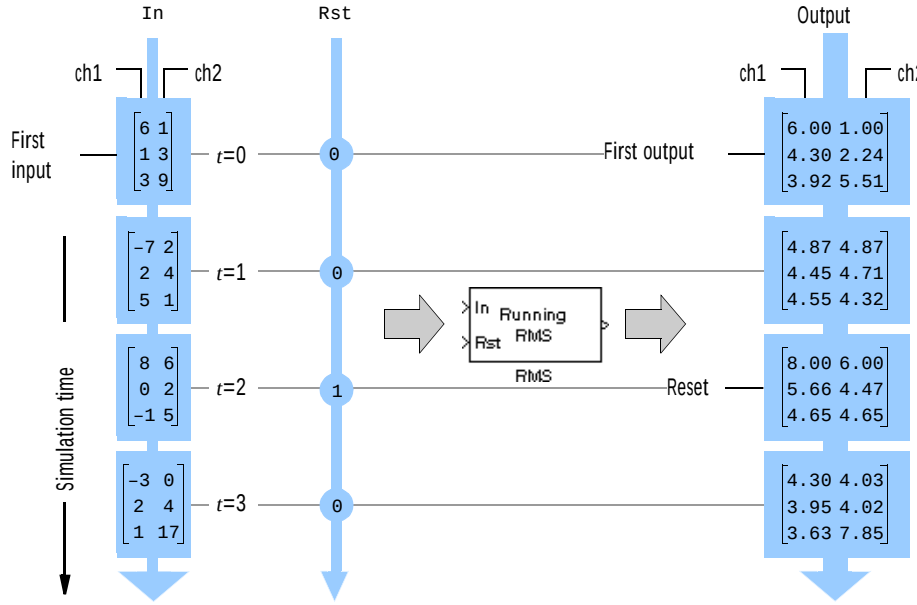
where

```
u = [6 1 3 -7 2 5 8 0 -1 -3 2 1;1 3 9 2 4 1 6 2 5 0 4 17]'
```

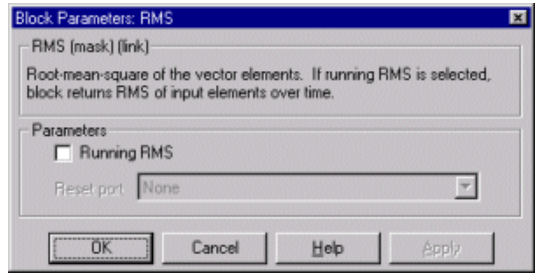
The Discrete Impulse block has the following settings:

- **Delay (samples)** = `2`
- **Sample time** = `1`
- **Samples per frame** = `1`

The block's operation is shown in the figure below.



Dialog Box



Running RMS

Enables running operation when selected.

Reset port

Enables the `Rst` input port when set to **Non-zero sample**, and disables the `Rst` input port when set to **None**.

See Also

Mean	DSP Blockset
Variance	DSP Blockset

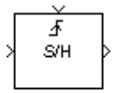
Purpose

Sample and hold an input signal.

Library

Signal Operations

Description



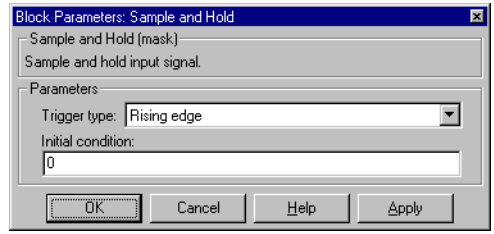
The Sample and Hold block acquires the input at the signal port whenever it receives a trigger event at the trigger port (marked by ⨍). The block then holds the output at the acquired input value until the next triggering event occurs. If the acquired input is frame-based, the output is frame-based; 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 zero to a positive value.
- **Falling edge** triggers the block to acquire the signal input when the trigger input falls from zero to a negative value.
- **Either edge** triggers the block to acquire the signal input when the trigger input either rises from zero to a positive value or falls from zero to a negative value.

The block's output prior to the first trigger event is specified by the **Initial condition** parameter. If the acquired input is an M-by-N matrix, the **Initial condition** can be an M-by-N matrix, or a scalar to be repeated across all elements of the matrix. If the input is a length-M 1-D vector, the **Initial condition** can be a length-M row or column vector, or a scalar to be repeated across all elements of the vector.

Dialog Box



Trigger type

The type of event that triggers the block to acquire the input signal.

Initial condition

The block's output prior to the first trigger event.

See Also

Downsample	DSP Blockset
N-Sample Switch	DSP Blockset

Purpose

Compute a nonparametric estimate of the spectrum using the short-time, fast Fourier transform (ST-FFT) method.

Library

Estimation / Power Spectrum Estimation

Description

The Short-Time FFT block computes a nonparametric estimate of the spectrum. The block averages the squared magnitude of the FFT computed over windowed sections of the input, and normalizes the spectral average by the square of the sum of the window samples.

Both an M-by-N frame-based matrix input and an M-by-N sample-based matrix input are treated as M sequential time samples from N independent channels. The block computes a separate estimate for each of the N independent channels and generates an N_{fft}-by-N matrix output. When **Inherit FFT length from input dimensions** is selected, N_{fft} is specified by the frame size of the input, which must be a power of 2. When **Inherit FFT length from input dimensions** is *not* selected, N_{fft} is specified as a power of 2 by the **FFT length** parameter, and the block zero pads or truncates the input to N_{fft} before computing the FFT.

Each column of the output matrix contains the estimate of the corresponding input column's power spectral density at N_{fft} equally spaced frequency points in the range $[0,F_s)$, where F_s is the signal's sample frequency. The output is always sample-based.

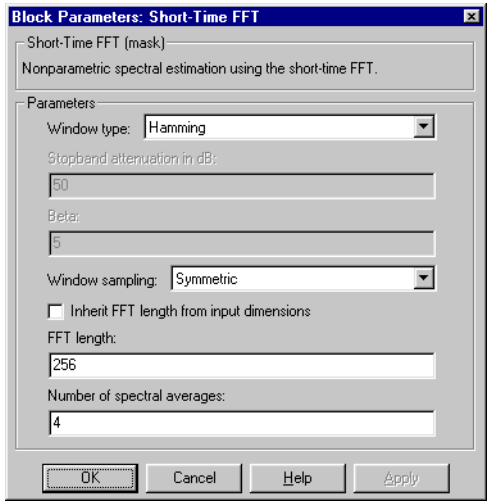
The **Number of spectral averages** specifies the number of spectra to average. Setting this parameter to `1` effectively disables averaging.

The **Window type**, **Stopband ripple**, **Beta**, and **Window sampling** parameters all apply to the specification of the window function; see the reference page for the Window Function block for more details on these four parameters.

Example

The `dspstfft` demo provides an illustration of using the Short-Time FFT and Matrix Viewer blocks to create a spectrogram. The `dspsacomp` demo compares the ST-FFT with several other spectral estimation methods.

Dialog Box



Window type

The type of window to apply. (See the Window Function block reference.) Tunable.

Stopband attenuation in dB

The level (dB) of stopband attenuation, R_s, for the Chebyshev window. Disabled for other **Window type** selections. Tunable.

Beta

The β parameter for the Kaiser window. Disabled for other **Window type** selections. Increasing **Beta** widens the mainlobe and decreases the amplitude of the window sidelobes in the window's frequency magnitude response. Tunable.

Window sampling

The window sampling, symmetric or periodic. Tunable.

Inherit FFT length from input dimensions

When selected, uses the input frame size as the number of data points, N_{fft}, on which to perform the FFT.

FFT length

The number of data points, N_{fft}, on which to perform the FFT. If N_{fft} exceeds the input frame size, the frame is zero-padded as needed. This

parameter is enabled when **Inherit FFT length from input dimensions** is not selected.

Number of spectral averages

The number of spectra to average; setting this parameter to `1` effectively disables averaging.

References

Oppenheim, A. V. and R. W. Schafer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989.

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

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 4-30 for related information.

Signal From Workspace

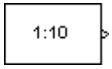
Purpose

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≠1), each of the N columns is treated as a distinct channel. The frame size is specified by the **Samples per frame** parameter, M_o, and the output is an M_o-by-N matrix containing M_o consecutive samples from each signal channel. The output sample period is specified by the **Sample time** parameter, T_s, and the output frame period is M_o*T_s. For M_o=1, the output is sample-based; otherwise the output is frame-based. For convenience, an imported row vector (M=1) is treated as a single channel, so the output dimension is M_o-by-1.

When the **Signal** parameter specifies an M-by-N-by-P array, each of the P pages (an M-by-N matrix) is output in sequence with period T_s. The **Samples per frame** parameter must be set to `1`, and the output is always sample-based.

Initial and Final Conditions

Unlike Simulink's From Workspace block, the Signal From Workspace block holds the output value constant between successive output frames (i.e., no linear interpolation takes place). Additionally, the initial signal values are always produced immediately at *t*=0.

When the block has output all of the available signal samples, it can start again at the beginning of the signal, or simply repeat the final value or generate zeros until the end of the simulation. (The block does not extrapolate the imported signal beyond the last sample.) The **Form output after final data value by** parameter controls this behavior:

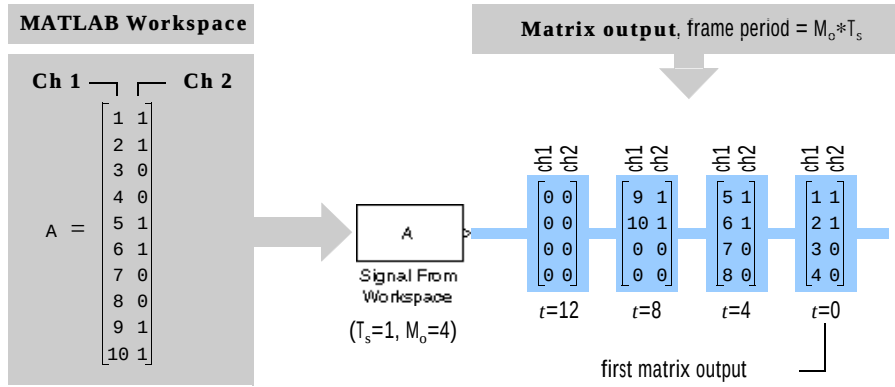
- If **Setting To Zero** is specified, the block generates zero-valued outputs for the duration of the simulation after generating the last frame of the signal.
- If **Holding Final Value** is specified, the block repeats the final sample for the duration of the simulation after generating the last frame of the signal.

- If **Cyclic Repetition** is specified, the block repeats the signal from the beginning after generating the last frame. If there are not enough samples at the end of the signal to fill the final frame, the block zero-pads the final frame as necessary to ensure that the output for each cycle is identical (e.g., the *i*th frame of one cycle contains the same samples as the *i*th frame of any other cycle).

Examples

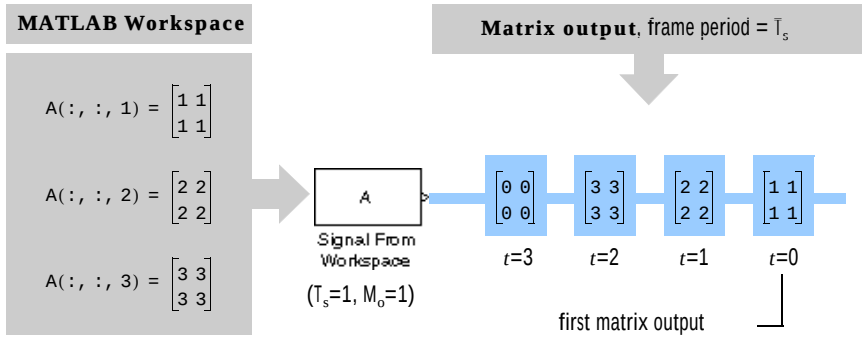
Example 1

In the first model below, the Signal From Workspace imports a two-channel signal from the workspace matrix `A`. The **Sample time** is set to `1` and the **Samples per frame** is set to 4, so the output is frame-based with a frame size of 4 and a frame period of 4 seconds. The **Form output after final data value by** parameter specifies **Setting To Zero**, so all outputs after the third frame (at *t*=8) are zero.



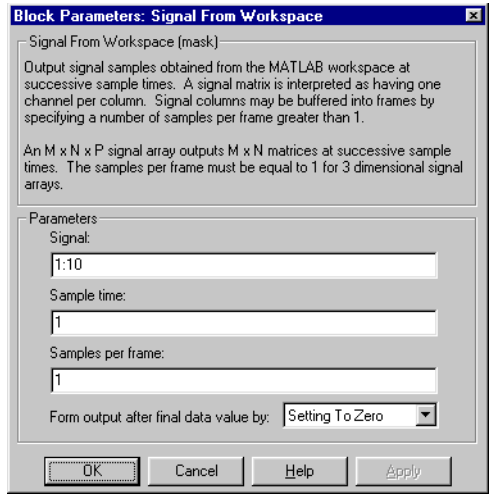
Example 2

In the second model below, the Signal From Workspace block imports a sample-based matrix signal from the 3-D workspace array `A`. Again, the **Form output after final data value by** parameter specifies **Setting To Zero**, so all outputs after the third (at *t*=2) are zero.



The **Samples per frame** parameter is set to `1` for 3-D input.

Dialog Box



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**).

See Also

From Wave Device	DSP Blockset
From Wave File	DSP Blockset
Sine Wave	DSP Blockset
To Workspace	Simulink
Triggered Signal From Workspace	DSP Blockset

See the sections below for related information:

- "Discrete-Time Signals" on page 3-3
- "Multichannel Signals" on page 3-11
- "Benefits of Frame-Based Processing" on page 3-14
- "Creating Signals Using the Signal From Workspace Block" on page 3-38
- "Importing Signals" on page 3-62

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

$$y = Ae^{j(2\pi ft + \phi)} = A\{\cos(2\pi ft + \phi) + j\sin(2\pi ft + \phi)\}$$

Each parameter value (A, f, ϕ) for real and complex sinusoids can be a scalar or length-N vector, where N is the desired number of channels in the output. If at least one parameter is specified as a length-N vector, scalar values specified for the other parameters are scalar expanded to length N (i.e., they are applied to every channel).

For example, specify

- **Amplitude** = `[1 2 3]`
- **Frequency** = `[1000 500 250]`
- **Phase** = `[0 0 pi/2]`

with **Output complexity** set to **Real** to generate a three-channel output containing the real sinusoids below.

$$y = \begin{cases} \sin(2000\pi t) & \text{(channel 1)} \\ 2\sin(1000\pi t) & \text{(channel 2)} \\ 3\sin\left(500\pi t + \frac{\pi}{2}\right) & \text{(channel 3)} \end{cases}$$

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.

The **Sample mode** parameter specifies the block's sampling property, which can be **Continuous** or **Discrete**, described below:

- **Continuous**

  In continuous mode, the sinusoid in the *i*th channel, y_i, is computed as a continuous function,

  $$y_i = A_i \sin(2\pi f_i t + \phi_i) \quad \text{(real)}$$

  or

  $$y_i = A_i e^{j(2\pi f_i t + \phi_i)} \quad \text{(complex)}$$

  and the block's output is continuous. In this mode, the block's operation is the same as that of a Simulink Sine Wave block with **Sample time** set to `0`. This mode offers high accuracy, but requires trigonometric function evaluations at each simulation step, which is computationally expensive. Additionally, because this method tracks absolute simulation time, a discontinuity will eventually occur when the time value reaches its maximum limit.

  Note also that many blocks in the DSP Blockset do not accept continuous-time inputs.

- **Discrete**

  In discrete mode, the block's discrete-time output can be generated by directly evaluating the trigonometric function, by table look-up, or by a differential method. The three options are explained below.

Discrete Computational Methods

When **Discrete** is selected from the **Sample mode** parameter, the secondary **Computation method** parameter provides three options for generating the discrete sinusoid:

- **Trigonometric Fcn**
- **Table Lookup**
- **Differential**

Trigonometric Fcn. The trigonometric function method computes the sinusoid in the ith channel, y_i, by sampling the continuous function

$$y_i = A_i \sin(2\pi f_i t + \phi_i) \qquad \text{(real)}$$

or

$$y_i = A_i e^{j(2\pi f_i t + \phi_i)} \qquad \text{(complex)}$$

with a period of T_s, where T_s is specified by the **Sample time** parameter. This mode of operation shares the same benefits and liabilities as the **Continuous** sample mode described above.

If the period of every sinusoid in the output is evenly divisible by the sample period, meaning that $1/(f_i T_s) = k_i$ 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, $1/(f_i T_s) = k_i$ must be an integer value for every channel i = 1, 2, ..., N. The table that is constructed for each channel contains k_i 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.

$$\sin(t + T_s) = \sin(t)\cos(T_s) + \cos(t)\sin(T_s)$$
$$\cos(t + T_s) = \cos(t)\cos(T_s) - \sin(t)\sin(T_s)$$

The update equations for the sinusoid in the ith channel, y_i, can therefore be written in matrix form (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 T_s is specified by the **Sample time** parameter. Since T_s is constant, the right-hand matrix is a constant and can be computed once at the start of the simulation. The value of $A_i\sin[2\pi f_i(t+T_s)+\phi_i]$ is then computed from the values of $\sin(2\pi f_i t+\phi_i)$ and $\cos(2\pi f_i t+\phi_i)$ by a simple matrix multiplication at each time step.

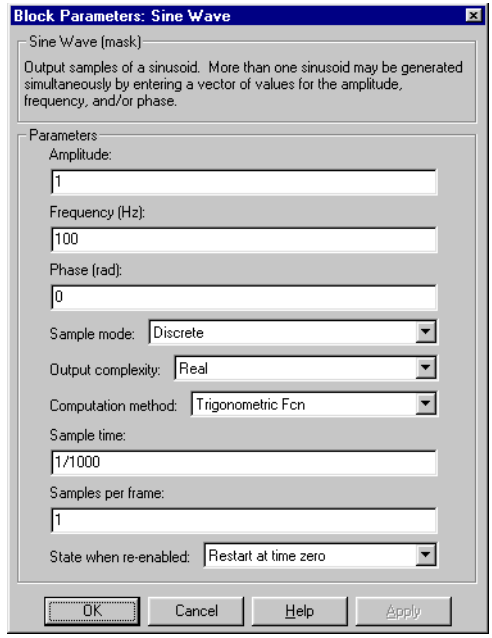
This mode offers reduced computational load, but is subject to drift over time due to cumulative quantization error. Because the method is not contingent on

an absolute time value, there is no danger of discontinuity during extended operations (when an absolute time variable might overflow).

Examples

The `dspsinecomp` demo provides a comparison of all the available sine generation methods.

Dialog Box



Amplitude

A length-N vector containing the amplitudes of the sine waves in each of N output channels, or a scalar to be applied to all N channels. The vector length must be the same as that specified for the **Frequency** and **Phase** parameters. Tunable; 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 the 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. Tunable; the frequency values can be altered while a

simulation is running, but the vector length must remain the same. Not tunable in Simulink's external mode when using the differential method.

Phase

A length-N vector containing the phase offsets, in radians, of the sine waves in each of N output channels, or a scalar to be applied to all N channels. The vector length must be the same as that specified for the **Amplitude** and **Frequency** parameters. Tunable; the phase values can be altered while a simulation is running, but the vector length must remain the same. Not tunable in Simulink's external mode when using the differential method.

Sample mode

The block's sampling behavior, **Continuous** or **Discrete**.

Output complexity

The type of waveform to generate: **Real** specifies a real sine wave, **Complex** specifies a complex exponential.

Computation method

The method by which discrete-time sinusoids are generated. This parameter is disabled when **Continuous** is selected from the **Sample mode** parameter.

Sample time

The period with which the sine wave is sampled, T_S. The block's output frame period is $M*T_S$, where M is specified by the **Samples per frame** parameter. This parameter is disabled when **Continuous** is selected from the **Sample mode** parameter.

Samples per frame

The number of consecutive samples from each sinusoid to buffer into the output frame, M. This parameter is disabled when **Continuous** is selected from the **Sample mode** parameter.

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.

See Also

Chirp	DSP Blockset
Complex Exponential	DSP Blockset
Signal From Workspace	DSP Blockset
Signal Generator	Simulink
Sine Wave	Simulink
`sin`	MATLAB

See “Creating Signals Using Signal Generator Blocks” on page 3-36 for related information.

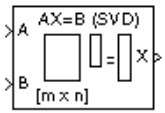
Purpose

Factor a matrix using singular value decomposition.

Library

Math Functions / Matrices and Linear Algebra / Matrix Factorizations

Description



The Singular Value Decomposition block factors the M-by-N input matrix A such that

$$A = U^{*}diag(S) \cdot V^{T}$$

where U is an M-by-P matrix, V is an N-by-P matrix, S is a length-P vector, and P is defined as min(M,N).

When M = N, U and V are both M-by-M unitary matrices. When M > N, V is an N-by-N unitary matrix, and U is an M-by-N matrix whose columns are the first N columns of a unitary matrix. When N > M, U is an M-by-M unitary matrix, and V is an M-by-N matrix whose columns are the first N columns of a unitary matrix. In all cases, S is a 1-D vector of positive singular values having length P. The output is always sample-based.

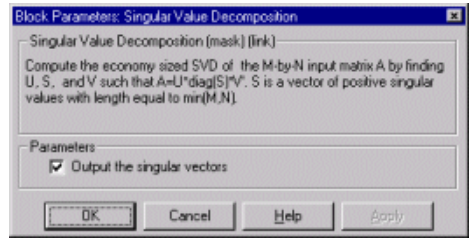
Length-N row inputs are treated as length-N columns.

```
[U,S,V] = svd(A,0)     % Equivalent MATLAB code for M > N
```

Note that the first (maximum) element of output S is equal to the 2-norm of the matrix A.

You can enable the `U` and `V` output ports by selecting the **Output the singular vectors** parameter.

Dialog Box



Compute singular vectors

Enables the `U` and `V` output ports when selected.

Singular Value Decomposition

References

Golub, G. H., and C. F. Van Loan. *Matrix Computations*. 3rd ed. Baltimore, MD: Johns Hopkins University Press, 1996.

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 4-32 for related information.

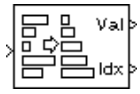
Purpose

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 *j*th column of `idx` is an index vector that permutes the *j*th column of u to the desired sorting order:

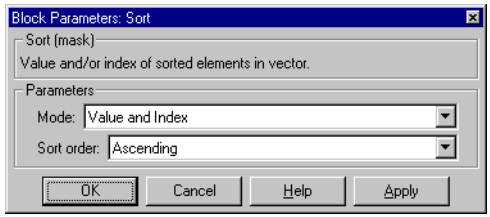
```
val(:,j) = u(idx(:,j),j)
```

As in **Value** mode, length-M 1-D vector inputs and *sample-based* length-M row vector inputs are both treated as M-by-1 column vectors.

Value and Index Mode

When **Mode** is set to **Value and Index**, the block outputs both the sorted matrix, `val`, and the index matrix, `idx`.

Dialog Box



Mode

The block's mode of operation: Output the sorted matrix (**Value**), the index matrix (**Index**), or both (**Value and Index**).

Sort order

The order in which to sort the input values, **Descending** or **Ascending**. Tunable, except in Simulink's external mode.

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 magnitude-squared FFT of the input, which can be a 1-D vector or a matrix of any frame status.

The Spectrum Scope block acquires a sequence of input samples into a buffer, and displays the squared magnitude of the FFT of each full buffer.

When the input is a 1-by-N sample-based vector or M-by-N sample-based matrix, you must select the **Buffer input** check box. Each of the N vector elements (or M∗N matrix elements) is then treated as an independent channel, and the block buffers and displays the data in each channel independently.

When the input is frame-based, you can leave the input as is, or rebuffer data by checking the **Buffer input** check box and specifying the new buffer size. In the latter case, you can also specify an optional **Buffer overlap**.

Buffering 1-D vector inputs is recommended. In this case, the inputs are buffered into frames (the length of which are specified in the **Buffer size** parameter), where each 1-D input vector becomes a row in the buffered outcome. If a 1-D vector input is left unbuffered, you will get a warning because the block is computing the FFT of a scalar; though the scope window appears, it is unlikely you will be able to see the plot, and a warning is also displayed on the scope itself. It is not recommended that you leave 1-D inputs unbuffered.

The number of input samples that the block buffers before computing and displaying the magnitude FFT is specified by the **Buffer size** parameter, M_o. The **Buffer overlap** parameter, L, specifies the number of samples from the previous buffer to include in the current buffer. The number of *new* input samples the block acquires before computing and displaying the magnitude FFT is the difference between the **Buffer size** and **Buffer overlap**, M_o-L.

The display update period is $(M_o\text{-}L)*T_s$, where T_s is the input sample period, and is *equal* to the input sample period when the **Buffer overlap** is M_o-1. For negative **Buffer overlap** values, the block simply discards the appropriate number of input samples after the buffer fills, and updates the scope display at a slower rate than the zero-overlap case.

When the **FFT length** check box is deselected and the input is buffered, the block uses the buffer size as the FFT size. If the check box is deselected and the input is not buffered, the block uses the input size as the FFT size. When the check box is selected, the **FFT length** parameter, N_{fft}, is enabled, and specifies the number of samples on which to perform the FFT. The block zero pads or truncates every channel's buffer to N_{fft} before computing the FFT.

The number of spectra to average is set by the **Number of spectral averages** parameter. Setting this parameter to 1 effectively disables averaging; See Short-Time FFT for more information.

In order to correctly scale the frequency axis (i.e., to determine the frequencies against which the transformed input data should be plotted), the block needs to know the actual sample period of the time-domain input. This is specified by the **Sample time of original time series** parameter, T_s.

When the **Inherit sample time from input** check box is selected, the block computes the frequency data from the sample period of the input to the block. This is valid when the following conditions hold:

- The input to the block is the original signal, with no samples added or deleted (by insertion of zeros, for example).
- The sample period of the time-domain signal in the simulation is equal to the period with which the physical signal was originally sampled.

One example when these conditions do not hold, is such as when the input to the block is not the original signal, but a zero-padded or otherwise rate-altered version. In such cases, you should specify the appropriate value for the **Sample time of original time-series** parameter.

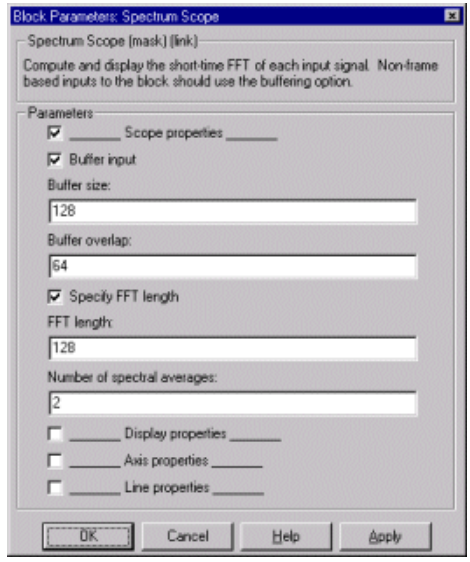
The **Frequency units** parameter specifies whether the frequency axis values should be in units of Hertz or rad/s, and the **Frequency range** parameter specifies the range of frequencies over which the magnitudes in the input should be plotted. The available options are **[0..Fs/2]**, **[-Fs/2..Fs/2]**, and **[0..Fs]**, where F_s is the time-domain signal's actual sample frequency. If the **Frequency units** parameter specifies **Hertz**, the spacing between frequency points is $1/(N_{fft}T_s)$. For **Frequency units** of **rad/sec**, the spacing between frequency points is $2\pi/(N_{fft}T_s)$.

Note that all of the FFT-based blocks in the DSP Blockset, including those in the Power Spectrum Estimation library, compute the FFT at frequencies in the

range $[0,F_s)$. The **Frequency range** parameter controls only the *displayed* range of the signal.

For information about the scope window, as well as the **Display properties**, **Axis properties**, and **Line properties** panels in the dialog box, see the reference page for the Vector Scope block.

Dialog Box



Scope properties

Select to expose **Scope properties** panel.

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 the number of spectra to average. Setting this parameter to `1` effectively disables averaging. See Short-Time FFT for more information.

Display properties

Select to expose the **Display properties** panel. See Vector Scope for more information. Tunable.

Axis properties

Select to expose the **Axis properties** panel. See Vector Scope for more information. Tunable.

Line properties

Select to expose the **Line properties** panel. See Vector Scope for more information. Tunable.

See Also

FFT	DSP Blockset
Vector Scope	DSP Blockset

See “Viewing Signals” on page 3-80 for related information.

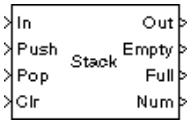
Purpose

Store inputs into a LIFO register.

Library

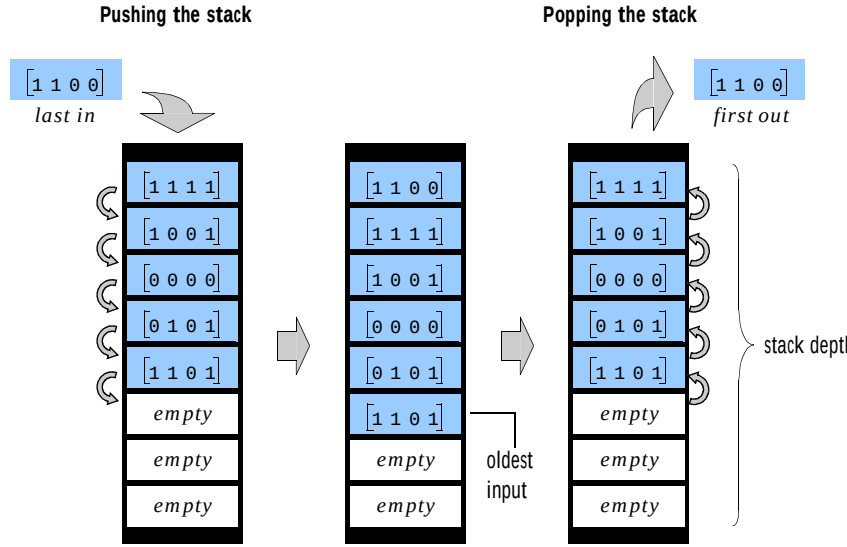
Signal Management / Buffers

Description



The Stack block stores a sequence of input samples in a LIFO (last in, first out) register. The register capacity is set by the **Stack depth** parameter, and inputs can be scalars, vectors, or matrices.

The block *pushes* the input at the `In` port onto the top of the stack when a trigger event is received at the `Push` port. When a trigger event is received at the `Pop` port, the block *pops* the top element off the stack and holds the `Out` port at that value. The last input to be pushed onto the stack is always the first to be popped off.



A trigger event at the optional `Clr` port (enabled by the **Clear input** check box) empties the stack contents. If **Clear output port on reset** is selected, then a trigger event at the `Clr` port empties the stack *and* sets the value at the `Out` port to zero. This setting also applies when a disabled subsystem containing the Stack block is re-enabled; the `Out` port value is only reset to zero in this case if **Clear output port on reset** is selected.

When two or more of the control input ports are triggered at the same time step, the operations are executed in the following order:

1. `Clr`
2. `Push`
3. `Pop`

The 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 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 **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.

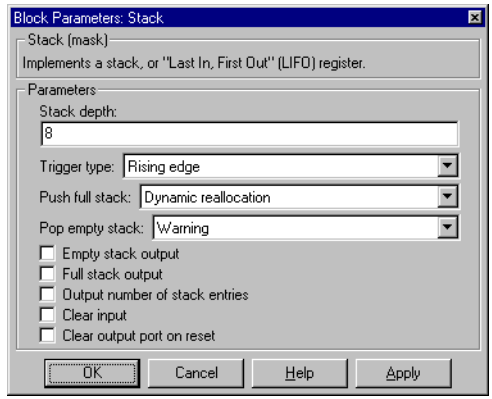
In	Push	Pop	Clr	Stack	Out	Empty	Full	Num
1	0	0	0	top [ ] [ ] [ ] [ ] bottom	0	1	0	0
2	1	0	0	top [2] [ ] [ ] [ ] bottom	0	0	0	1
3	0	0	0	top [3] [2] [ ] [ ] bottom	0	0	0	2
4	1	0	0	top [4] [3] [2] [ ] bottom	0	0	0	3
5	0	0	0	top [5] [4] [3] [2] bottom	0	0	1	4
6	0	1	0	top [4] [3] [2] [ ] bottom	5	0	0	3
7	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
10	1	0	0	top [10] [ ] [ ] [ ] bottom	2	0	0	1
11	0	0	0	top [11] [10] [ ] [ ] bottom	2	0	0	2
12	1	0	1	top [12] [ ] [ ] [ ] bottom	0	0	0	1

Note that at the last step shown, the `Push` and `Clr` ports are triggered simultaneously. The `Clr` trigger takes precedence, and the stack is first cleared and then pushed.

Example 2

The `dspqdemo` demo provides an example of the related Queue block.

Dialog Box



Stack depth

The number of entries that the LIFO register can hold.

Trigger type

The type of event that triggers the block's execution.

Push full stack

Response to a trigger received at the `Push` port when the register is full.

Pop empty stack

Response to a trigger received at the `Pop` port when the register is empty. 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.

Clear input

Enable the `Clr` input port, which empties the stack when the trigger specified by the **Trigger type** is received.

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.

See Also

Buffer	DSP Blockset
Delay Line	DSP Blockset
Queue	DSP Blockset

Standard Deviation

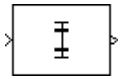
Purpose

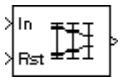
Find the standard deviation of an input or sequence of inputs.

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

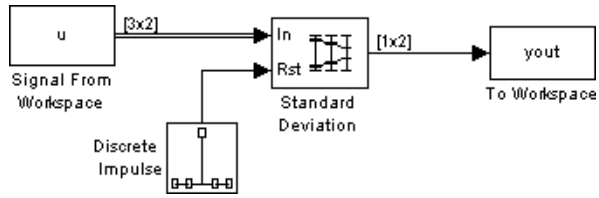
When the **Running standard deviation** check box is selected, the block tracks the standard deviation of each channel in a *time-sequence* of M-by-N inputs. For sample-based inputs, the output is a sample-based M-by-N matrix with each element y_{ij} containing the standard deviation of element u_{ij} 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.

If the **Reset port** parameter is set to **Non-zero sample**, the optional `Rst` port is enabled and the block resets the running standard deviation when the scalar input at the `Rst` port is nonzero. (The `Rst` port can be disabled by setting the **Reset port** parameter to **None**.)

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.

Example

The Standard Deviation block in the model below calculates the running standard deviation of a frame-based 3-by-2 (two-channel) matrix input, `u`. The running standard deviation is reset at t=2 by an impulse to the block's `Rst` port.



The Standard Deviation block has the following settings:

- **Running standard deviation** = ☑
- **Reset port** = **Non-zero sample**

The Signal From Workspace block has the following settings:

- **Signal** = `u`
- **Sample time** = `1/3`
- **Samples per frame** = `3`

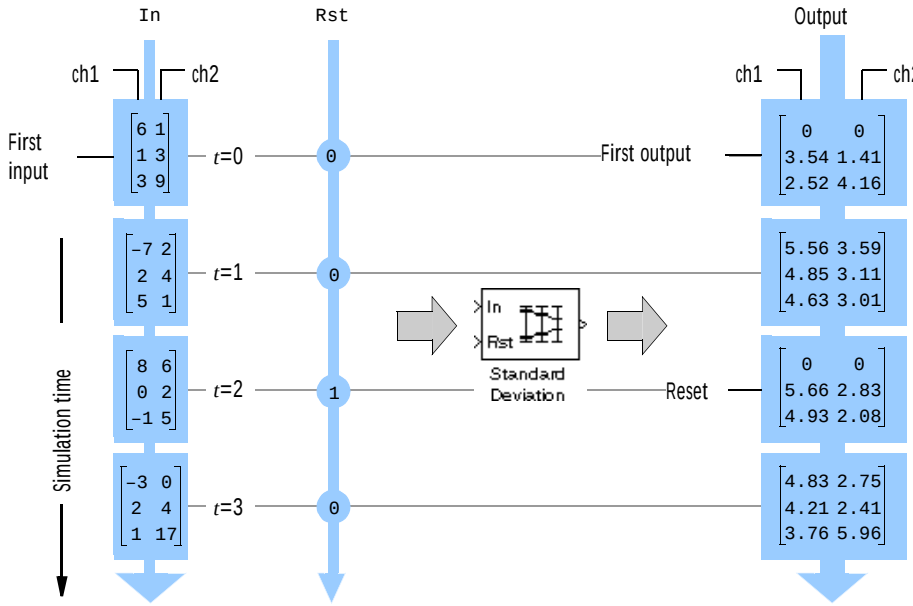
where

```
u = [6 1 3 -7 2 5 8 0 -1 -3 2 1;1 3 9 2 4 1 6 2 5 0 4 17]'
```

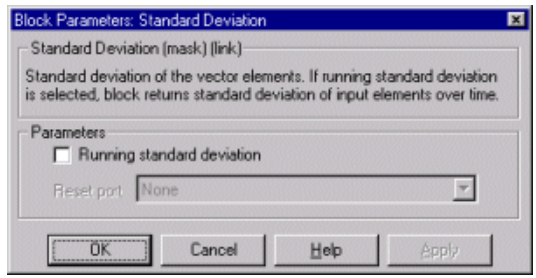
The Discrete Impulse block has the following settings:

- **Delay (samples)** = 2
- **Sample time** = 1
- **Samples per frame** = 1

The block's operation is shown in the figure below.



Dialog Box



Running standard deviation

Enables running operation when selected.

Reset port

Enables the `Rst` input port when set to **Non-zero sample**, and disables the `Rst` input port when set to **None**.

See Also

Mean	DSP Blockset
RMS	DSP Blockset
Variance	DSP Blockset
`std`	MATLAB

Submatrix

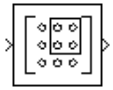
Purpose

Select a subset of elements (submatrix) from a matrix input.

Library

Math Functions / Matrices and Linear Algebra / Matrix Operations,
Signal Management / Indexing

Description



The Submatrix block extracts a contiguous submatrix from the M-by-N input matrix u. A length-M 1-D vector input is treated as an M-by-1 matrix. The **Row span** parameter provides three options for specifying the range of rows in u to be retained in submatrix output y:

- **All rows**

  Specifies that y contains all M rows of u.

- **One row**

  Specifies that y contains only one row from u. The **Starting row** parameter (described below) is enabled to allow selection of the desired row.

- **Range of rows**

  Specifies that y contains one or more rows from u. The **Row** and **Ending row** parameters (described below) are enabled to allow selection of the desired range of rows.

The **Column span** parameter contains a corresponding set of three options for specifying the range of columns in u to be retained in submatrix y: **All columns**, **One column**, or **Range of columns**. The **One column** option enables the **Column** parameter, and **Range of columns** options enable the **Starting column** and **Ending column** parameters.

The output has the same frame status as the input.

Range Specification Options

When **One row** or **Range of rows** is selected from the **Row span** parameter, the desired row or range of rows is specified by the **Row** parameter, or the **Starting row** and **Ending row** parameters. Similarly, when **One column** or **Range of columns** is selected from the **Column span** parameter, the desired column or range of columns is specified by the **Column** parameter, or the **Starting column** and **Ending column** parameters.

The **Row**, **Column**, **Starting row** or **Starting column** can be specified in six ways:

- **First**

  For rows, this specifies that the first row of u should be used as the first row of y. If all columns are to be included, this is equivalent to `y(1,:) = u(1,:)`.

  For columns, this specifies that the first column of u should be used as the first column of y. If all rows are to be included, this is equivalent to `y(:,1) = u(:,1)`.

- **Index**

  For rows, this specifies that the row of u, `firstrow`, forward-indexed by the **Row index** parameter or the **Starting row index** parameter, should be used as the first row of y. If all columns are to be included, this is equivalent to `y(1,:) = u(firstrow,:)`.

  For columns, this specifies that the column of u, forward-indexed by the **Column index** parameter or the **Starting column index** parameter, `firstcol`, should be used as the first column of y. If all rows are to be included, this is equivalent to `y(:,1) = u(:,firstcol)`.

- **Offset from last**

  For rows, this specifies that the row of u offset from row M by the **Row offset** or **Starting row offset** parameter, `firstrow`, should be used as the first row of y. If all columns are to be included, this is equivalent to `y(1,:) = u(M-firstrow,:)`.

  For columns, this specifies that the column of u offset from column N by the **Column offset** or **Starting column offset** parameter, `firstcol`, should be used as the first column of y. If all rows are to be included, this is equivalent to `y(:,1) = u(:,N-firstcol)`.

- **Last**

  For rows, this specifies that the last row of u should be used as the only row of y. If all columns are to be included, this is equivalent to `y = u(M,:)`.

  For columns, this specifies that the last column of u should be used as the only column of y. If all rows are to be included, this is equivalent to `y = u(:,N)`.

- **Offset from middle**

  For rows, this specifies that the row of u offset from row M/2 by the **Starting row offset** parameter, `firstrow`, should be used as the first row of y. If all

columns are to be included, this is equivalent to `y(1,:) = u(M/2-firstrow,:)`.

For columns, this specifies that the column of u offset from column N/2 by the **Starting column offset** parameter, `firstcol`, should be used as the first column of y. If all rows are to be included, this is equivalent to `y(:,1) = u(:,N/2-firstcol)`.

- **Middle**

  For rows, this specifies that the middle row of u should be used as the only row of y. If all columns are to be included, this is equivalent to `y = u(M/2,:)`.

  For columns, this specifies that the middle column of u should be used as the only column of y. If all rows are to be included, this is equivalent to `y = u(:,N/2)`.

The **Ending row** or **Ending column** can similarly be specified in five ways:

- **Index**

  For rows, this specifies that the row of u forward-indexed by the **Ending row index** parameter, `lastrow`, should be used as the last row of y. If all columns are to be included, this is equivalent to `y(end,:) = u(lastrow,:)`.

  For columns, this specifies that the column of u forward-indexed by the **Ending column index** parameter, `lastcol`, should be used as the last column of y. If all rows are to be included, this is equivalent to `y(:,end) = u(:,lastcol)`.

- **Offset from last**

  For rows, this specifies that the row of u offset from row M by the **Ending row offset** parameter, `lastrow`, should be used as the last row of y. If all columns are to be included, this is equivalent to `y(end,:) = u(M-lastrow,:)`.

  For columns, this specifies that the column of u offset from column N by the **Ending column offset** parameter, `lastcol`, should be used as the last column of y. If all rows are to be included, this is equivalent to `y(:,end) = u(:,N-lastcol)`.

- **Last**

  For rows, this specifies that the last row of u should be used as the last row of y. If all columns are to be included, this is equivalent to `y(end,:) = u(M,:)`.

  For columns, this specifies that the last column of u should be used as the last column of y. If all rows are to be included, this is equivalent to `y(:,end) = u(:,N)`.

- **Offset from middle**

  For rows, this specifies that the row of u offset from row M/2 by the **Ending row offset** parameter, `lastrow`, should be used as the last row of y. If all columns are to be included, this is equivalent to `y(end,:) = u(M/2-lastrow,:)`.

  For columns, this specifies that the column of u offset from column N/2 by the **Ending column offset** parameter, `lastcol`, should be used as the last column of y. If all rows are to be included, this is equivalent to `y(:,end) = u(:,N/2-lastcol)`.

- **Middle**

  For rows, this specifies that the middle row of u should be used as the last row of y. If all columns are to be included, this is equivalent to `y(end,:) = u(M/2,:)`.

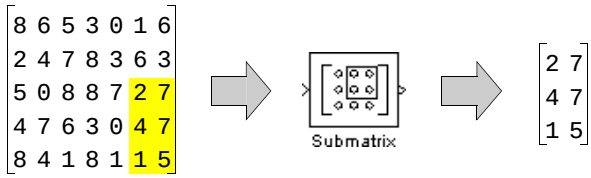
  For columns, this specifies that the middle column of u should be used as the last column of y. If all rows are to be included, this is equivalent to `y(:,end) = u(:,N/2)`.

Example

To extract the lower-right 3-by-2 submatrix from a 5-by-7 input matrix, enter the following set of parameters:

- **Row span** = **Range of rows**
- **Starting row** = **Index**
- **Starting row index** = `3`
- **Ending row** = **Last**
- **Column span** = **Range of columns**
- **Starting column** = **Offset from last**
- **Starting column offset** = `1`
- **Ending column** = **Last**

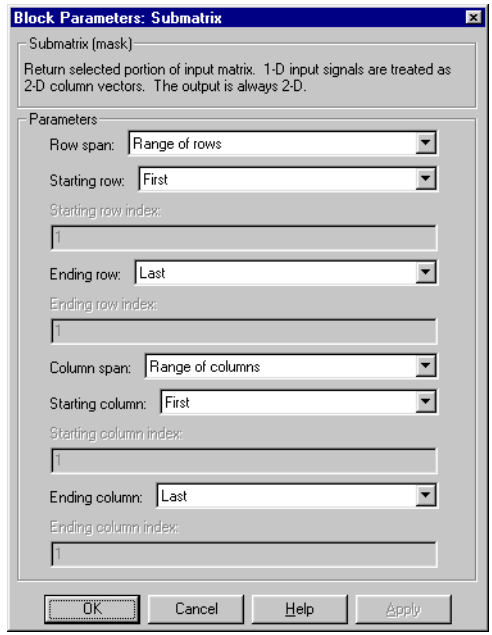
The figure below shows the operation for a 5-by-7 matrix with random integer elements, `randint(5,7,10)`.



There are often several possible parameter combinations that select the *same* submatrix from the input. For example, instead of specifying **Last** for **Ending column**, you could select the same submatrix by specifying:

- **Ending column** = **Index**
- **Ending column index** = 7

Dialog Box



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

Row span

The range of input rows to be retained in the output. Options are **All rows**, **One row**, or **Range of rows**.

Row/Starting row

The input row to be used as the first row of the output. **Row** is enabled when **One row** is selected from **Row span**, and **Starting row** when **Range of rows** is selected from **Row span**.

Row index/Starting row index

The index of the input row to be used as the first row of the output. **Row index** is enabled when **Index** is selected from Row, and **Starting row index** when **Index** is selected from **Starting row**.

Row offset/Starting row offset

The offset of the input row to be used as the first row of the output. **Row offset** is enabled when **Offset from middle** or **Offset from last** is selected from **Row**, and Starting row offset is enabled when **Offset from middle** or **Offset from last** is selected from **Starting row**.

Ending row

The input row to be used as the last row of the output. This parameter is enabled when **Range of rows** is selected from **Row span** and any option but **Last** is selected from **Starting row**.

Ending row index

The index of the input row to be used as the last row of the output. This parameter is enabled when **Index** is selected from **Ending row**.

Ending row offset

The offset of the input row to be used as the last row of the output. This parameter is enabled when **Offset from middle** or **Offset from last** is selected from **Ending row**.

Column span

The range of input columns to be retained in the output. Options are **All columns**, **One column**, or **Range of columns**.

Column/Starting column

The input column to be used as the first column of the output. **Column** is enabled when **One column** is selected from **Column span**, and **Starting column** is enabled when **Range of columns** is selected from **Column span**.

Column index/Starting column index

The index of the input column to be used as the first column of the output. **Column index** is enabled when **Index** is selected from Column, and **Starting column index** is enabled when **Index** is selected from **Starting column**.

Column offset/Starting column offset

The offset of the input column to be used as the first column of the output. **Column offset** is enabled when **Offset from middle** or **Offset from last** is selected from Column. **Starting column offset** is enabled when **Offset from middle** or **Offset from last** is selected from **Starting column**.

Ending column

The input column to be used as the last column of the output. This parameter is enabled when **Range of columns** is selected from **Column span** and any option but **Last** is selected from **Starting column**.

Ending column index

The index of the input column to be used as the last column of the output. This parameter is enabled when **Index** is selected from **Ending column**.

Ending column offset

The offset of the input column to be used as the last column of the output. This parameter is enabled when **Offset from middle** or **Offset from last** is selected from **Ending column**.

See Also

Reshape	Simulink
Selector	Simulink
Variable Selector	DSP Blockset
`reshape`	MATLAB

See “Deconstructing Signals” on page 3-54 for related information.

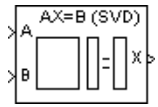
Purpose

Solve the equation A*X*=B using singular value decomposition.

Library

Math Functions / Matrices and Linear Algebra / Linear System Solvers

Description

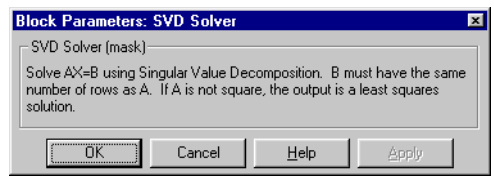


The SVD Solver block solves the linear system A*X*=B, which can be overdetermined, underdetermined, or exactly determined. The system is solved by applying SVD factorization to the M-by-N matrix, A, at the A port. The input to the B port is the right hand-side M-by-L matrix, B. A length-M 1-D vector input at either port is treated as an M-by-1 matrix.

The output at the x port is the N-by-L matrix, *X*. *X* is always sample based, and is chosen to minimize the sum of the squares of the elements of B-A*X*. When B is a vector, this solution minimizes the vector 2-norm of the residual (B-A*X* 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 A*X*=B. The minimum-norm-residual solution is unique for overdetermined and exactly determined linear systems, but it is not unique for underdetermined linear systems. Thus when the SVD Solver is applied to an underdetermined system, the output *X* is chosen such that the number of nonzero entries in *X* is minimized.

Dialog Box



SVD Solver

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 4-31 for related information.

Time-Varying Direct-Form II Transpose Filter

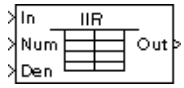
Purpose

Apply a variable IIR filter to the input.

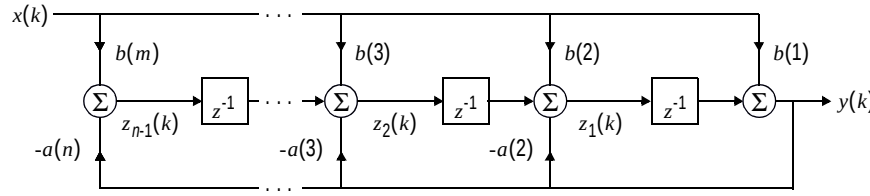
Library

Filtering / Filter Structures

Description



The Time-Varying Direct-Form II Transpose Filter block is a version of the Direct-Form II Transpose Filter block whose filter coefficients can be updated during the simulation. The block applies a direct-form II transposed IIR filter to the top input (`In`).



This is a canonical form that has the minimum number of delay elements. The filter order is `max(m,n)-1`.

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

The block's two lower inputs (`Num` and `Den`) specify the filter's transfer function,

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_1 + b_2 z^{-1} + \ldots + b_{m+1} z^{-(m-1)}}{a_1 + a_2 z^{-1} + \ldots + a_{n+1} z^{-(n-1)}}$$

By default the filter coefficients are normalized by a_1. To prevent normalization by a_1, deselect the **Support non-normalized filters** check box.

Filter Type

The **Filter type** parameter specifies whether the filter is an all-zero (FIR or MA) filter, all-pole (AR) filter, or pole-zero (IIR or ARMA) filter:

- **Pole-zero**

  The block accepts inputs for both the numerator (`Num`) and denominator (`Den`) vectors.

  Input `Num` is a vector of numerator coefficients,

  ```
  [b(1) b(2) ... b(m)]
  ```

  and input `Den` is a vector of denominator coefficients,

  ```
  [a(1) a(2) ... a(n)]
  ```

- **All-zero**

  The block accepts only the numerator vector (`Num`). The denominator of the all-zero filter is 1.

- **All-pole**

  The block accepts only the denominator vector (`Den`). The numerator of the all-pole filter is 1.

For any of these designs, the coefficient vector inputs can change over time to alter the filter's response characteristics during the simulation.

Initial Conditions

In its default form, the filter initializes the internal filter states to zero, which is equivalent to assuming past inputs and outputs are zero. The block also accepts optional nonzero initial conditions for the filter delays. Note that the number of filter states (delay elements) per input channel is

```
max(m,n)-1
```

The **Initial conditions** parameter may take one of four forms:

- Empty matrix

  The empty matrix, `[]`, causes a zero (0) initial condition to be applied to all delay elements in each filter channel.

- Scalar

  The scalar value is copied to all delay elements in each filter channel. Note that a value of zero is equivalent to setting the **Initial conditions** parameter to the empty matrix, `[]`.

- Vector

  The vector has a length equal to the number of delay elements in each filter channel, `max(m,n)-1`, and specifies a unique initial condition for each delay element in the filter channel. This vector of initial conditions is applied to each filter channel.

- Matrix

  The matrix specifies a unique initial condition for each delay element, and can specify different initial conditions for each filter channel. The matrix must have the same number of rows as the number of delay elements in the filter, `max(m,n)-1`, and must have one column per filter channel.

Filter Update Rate

In frame-based operation, the **Filter update rate** parameter determines how frequently the block updates the filter coefficients (i.e., how often it checks the `Num` and `Den` inputs). There are two available options:

- **One filter per sample time**

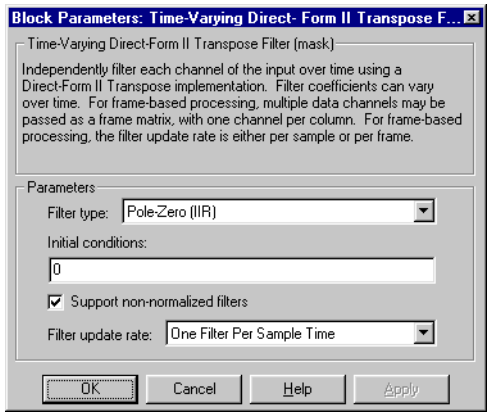
  The block updates the filter coefficients (from inputs `Num` and `Den`) for each individual scalar sample in the frame-based input. This means that each output sample could potentially be computed by a different filter (assuming that `Num` and `Den` inputs are updated frequently enough).

- **One filter per frame time**

  The block updates the filter coefficients (from inputs `Num` and `Den`) for each new input frame, rather than at each sample in the frame. This means that each output sample in a given frame is a result of an identical filtering process.

Time-Varying Direct-Form II Transpose Filter

Dialog Box



Filter type

The type of filter to apply: **Pole-Zero (IIR)**, **All-Zero (FIR)**, or **All-Pole (AR)**. The `Num` and `Den` input ports are enabled or disabled as appropriate.

Initial conditions

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

Support non-normalized filters

Normalizes the filter by a_1 when selected.

Filter update rate

The frequency with which the block updates the filter coefficients; once per sample, or once per frame.

References

Oppenheim, A. V. and R. W. Schafer. *Discrete-Time Signal Processing.* Englewood Cliffs, NJ: Prentice Hall, 1989.

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

See Also

Discrete Filter	Simulink
Filter Realization Wizard	DSP Blockset
Direct-Form II Transpose Filter	DSP Blockset
Time-Varying Lattice Filter	DSP Blockset
`filter`	MATLAB

See “Filter Structures” on page 4-23 for related information.

Time-Varying Lattice Filter

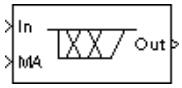
Purpose

Apply a variable lattice filter to the input.

Library

Filtering / Filter Structures

Description



The Time-Varying Lattice Filter block applies a moving average (MA) or autoregressive (AR) lattice filter to the top input (`In`). The filter reflection coefficients are specified by the vector input to the `MA` or `AR` port, and can vary with time.

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

Filter Type

The **Filter type** parameter specifies whether the filter is an all-zero (FIR or MA) filter or all-pole (AR) filter.

- **All-zero**

  The block constructs an *n*th order MA filter using the *n* reflection coefficients contained in the vector input to the `MA` port.

  ```
  k = [k(1) k(2) ... k(n)]
  ```

- **All-pole**

  The block constructs an *n*th order AR filter using the *n* reflection coefficients contained in the vector input to the `AR` port.

  ```
  k = [k(1) k(2) ... k(n)]
  ```

For both designs, the coefficient vector inputs can change over time to alter the filter's response characteristics during the simulation.

Initial Conditions

In its default form, the filter initializes the internal filter states to zero, which is equivalent to assuming past inputs and outputs are zero. The block also accepts optional nonzero initial conditions for the filter delays. Note that the number of filter states (delay elements) per input channel is

```
length(k)
```

The **Initial conditions** parameter may take one of four forms:

- Empty matrix

  The empty matrix, `[]`, causes a zero (0) initial condition to be applied to all delay elements in each filter channel.

- Scalar

  The scalar value is copied to all delay elements in each filter channel. Note that a value of zero is equivalent to setting the **Initial conditions** parameter to the empty matrix.

- Vector

  The vector has a length equal to the number of delay elements in each filter channel, `length(k)`, and specifies a unique initial condition for each delay element in the filter channel. This vector of initial conditions is applied to each filter channel.

- Matrix

  The matrix specifies a unique initial condition for each delay element, and can specify different initial conditions for each filter channel. The matrix must have the same number of rows as the number of delay elements in the filter, `length(k)`, and must have one column per filter channel.

Filter Update Rate

In frame-based operation, the **Filter update rate** parameter determines how frequently the block updates the filter coefficients (i.e., how often it checks the `MA` or `AR` input). There are two available options:

- **One filter per sample time**

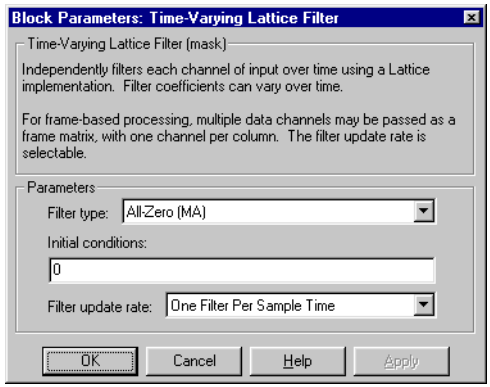
  The block updates the filter coefficients (from input `MA` or `AR`) for each individual scalar sample in the framed input. This means that each output sample could potentially be computed by a different filter (assuming that the `MA` or `AR` input is updated frequently enough).

- **One filter per frame time**

  The block updates the filter coefficients (from input `MA` or `AR`) for each new input frame, rather than at each sample in the frame. This means that each output sample in a given frame is a result of an identical filtering process.

Time-Varying Lattice Filter

Dialog Box



Filter type

The type of filter to apply: MA or AR. The `MA` or `AR` input port is enabled or disabled appropriately.

Initial conditions

The filter's initial conditions.

Filter update rate

The frequency with which the block updates the filter coefficients; once per sample, or once per frame.

References

Oppenheim, A. V. and R. W. Schafer. *Discrete-Time Signal Processing.* Englewood Cliffs, NJ: Prentice Hall, 1989.

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

See Also

Discrete Filter	Simulink
Direct-Form II Transpose Filter	DSP Blockset
Filter Realization Wizard	DSP Blockset
Time-Varying Direct-Form II Transpose Filter	DSP Blockset
`filter`	MATLAB

See "Filter Structures" on page 4-23 for related information.

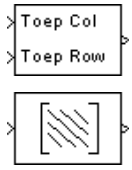
Purpose

Generate a matrix with Toeplitz symmetry.

Library

Math Functions / Matrices and Linear Algebra / Matrix Operations

Description



The Toeplitz block generates a Toeplitz matrix from inputs defining the first column and first row. The top input (`Col`) is a vector containing the values to be placed in the first *column* of the matrix, and the bottom input (`Row`) is a vector containing the values to be placed in the first *row* of the matrix.

```
y = toeplitz(Col,Row)      % Equivalent MATLAB code
```

The other elements of the matrix obey the relationship

```
y(i,j) = y(i-1,j-1)
```

and the output has dimension `[length(Col) length(Row)]`. The `y(1,1)` element is inherited from the `Col` input. For example, the following inputs

```
Col = [1 2 3 4 5]
Row = [7 7 3 3 2 1 3]
```

produce the Toeplitz matrix

$$\begin{bmatrix} 1 & 7 & 3 & 3 & 2 & 1 & 3 \\ 2 & 1 & 7 & 3 & 3 & 2 & 1 \\ 3 & 2 & 1 & 7 & 3 & 3 & 2 \\ 4 & 3 & 2 & 1 & 7 & 3 & 3 \\ 5 & 4 & 3 & 2 & 1 & 7 & 3 \end{bmatrix}$$

If both of the inputs are sample-based, the output is sample-based. Otherwise, the output is frame-based.

When the **Symmetric** check box is selected, the block generates a symmetric (Hermitian) Toeplitz matrix from a single input, `u`, defining both the first row and first column of the matrix.

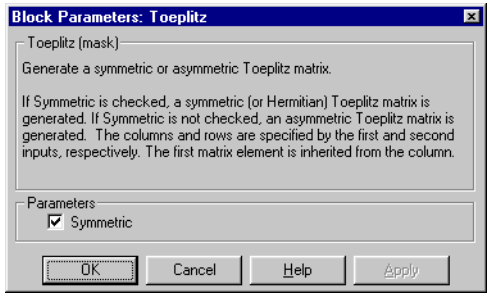
```
y = toeplitz(u)        % Equivalent MATLAB code
```

The output has dimension `[length(u) length(u)]`. For example, the Toeplitz matrix generated from the input vector `[1 2 3 4]` is

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

The output has the same frame status as the input.

Dialog Box



Symmetric

When selected, enables the single-input configuration for symmetric Toeplitz matrix output.

See Also

Constant Diagonal Matrix
`toeplitz`
DSP Blockset
MATLAB

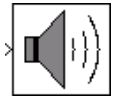
Purpose

Send audio data to a standard audio device in real-time (Windows only).

Library

DSP Sinks

Description



The To Wave Device block sends audio data to a standard Windows audio device in real-time. It is compatible with most popular Windows hardware, including Sound Blaster cards. (Models that contain both this block and the From Wave Device block require a *duplex-capable* sound card.) The data is sent to the hardware in uncompressed PCM (pulse code modulation) format, and should typically be sampled at one of the standard Windows audio device rates: 8000, 11025, 22050, or 44100 Hz. Some hardware may support other rates in addition to these.

The **Use default audio device** parameter allows the block to detect and use the system's default audio hardware. This option should be selected on systems that have a single sound device installed, or when the default sound device on a multiple-device system is the desired target. In cases when the default sound device is *not* the desired output device, deselect **Use default audio device**, and enter the desired device identification number in the **Audio device ID** parameter. The device ID is an integer value that the block associates with the sound device. A 3-device system, for example, has device ID numbers of `1`, `2`, and `3`.

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 `double`, `single`, or `int16`, the block conveys the signal samples to the audio device using 16 bits. If the input data type is `uint8`, the block conveys the signal samples to the audio device using 8 bits.

```
sound(u,Fs,bits)       % Equivalent MATLAB code
```

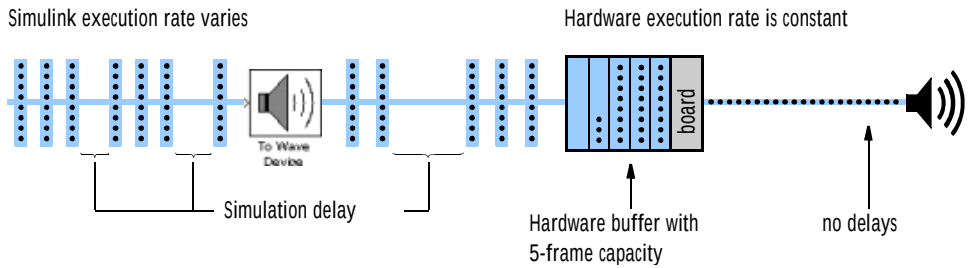
Note that the block does not support `uint16` or `int8` data types. The 16-bit sample width requires more memory but in general yields better fidelity. The amplitude of the input should be in the range ±1. Values outside this range are clipped to the nearest allowable value.

Buffering

Because the audio device generates real-time audio output, Simulink must maintain a continuous flow of data to the device throughout the simulation. Delays in passing data to the audio hardware can result in hardware errors or distortion of the output. This means that the To Wave Device block must in principle supply data to the audio hardware as quickly as the hardware reads the data. However, the To Wave Device block often *cannot* match the throughput rate of the audio hardware, especially when the simulation is running from within Simulink rather than as generated code. (Simulink execution speed routinely varies during the simulation as the host operating system services other processes.) The block must therefore rely on a buffering strategy to ensure that signal data is accessible to the hardware on demand.

At the start of the simulation, the To Wave Device block writes T_d seconds worth of signal data to the device (hardware) buffer, where T_d is specified by the **Initial output delay** parameter. When this initial data is loaded into the buffer, the audio device begins processing the buffered data, and continues at a constant rate until the buffer empties. The size of the buffer, T_b, is specified by the **Queue duration** parameter. As the audio device reads data from the *front* of the buffer, the To Wave Device block continues appending inputs to the back of the buffer at the rate they are received.

The following figure shows an audio signal with 8 samples per frame. The buffer of the sound board has a five-frame capacity, not fully used at the instant shown. (If the signal sample rate was 8kHz, for instance, this small buffer could hold approximately 0.005 seconds of data.)



If the simulation throughput rate is higher than the hardware throughput rate, the buffer remains at a constant level throughout the simulation. If necessary, the To Wave Device block buffers inputs until space becomes available in the hardware buffer (i.e., data is not thrown away). More typically,

the hardware throughput rate is higher than the simulation throughput rate, and the buffer tends to empty over the duration of the simulation.

Under normal operation, an empty buffer indicates that the simulation is finished, and the entire length of the audio signal has been processed. However, if the buffer size is too small in relation to the simulation throughput rate, the buffer may also empty before the entire length of signal is processed. This usually results in a device error or undesired device output.

When the device fails to process the entire signal length because the buffer prematurely empties, you can choose to either increase the buffer size or the simulation throughput rate.

- *Increase the buffer size.* The **Queue duration** parameter specifies the length of signal, T_b (in real-time seconds), to buffer to the audio device during the simulation. The number of frames buffered is approximately

  $$\frac{T_b F_s}{M_o}$$

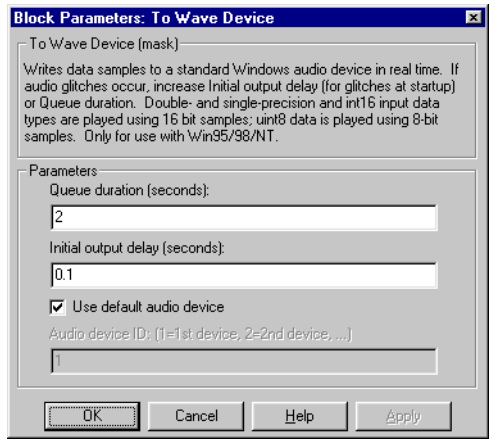
  where F_s is the sample rate of the signal and M_o is the number of samples per frame. The optimal buffer size for a given signal depends on the signal length, the frame size, and the speed of the simulation. The maximum number of frames that can be buffered is 1024.
- *Increase the simulation throughput rate.* Two useful methods for improving simulation throughput rates are increasing the signal frame size and compiling the simulation into native code.
  - Increase frame sizes (and convert sample-based signals to frame-based signals) throughout the model to reduce the amount of block-to-block communication overhead. This can drastically increase throughput rates in many cases. However, larger frame sizes generally result in greater model latency due to initial buffering operations. (Note that increasing the audio signal frame size does not affect the number of samples buffered to the hardware since the **Queue duration** is specified in seconds.)
  - Generate executable code with Real-Time Workshop. Native code runs much faster than Simulink, and should provide rates adequate for real-time audio processing.

Audio problems at startup can often be corrected by entering a larger value for the **Initial output delay** parameter, which allows a greater portion of the

signal to be preloaded into the hardware buffer. A value of `0` for the **Initial output delay** parameter specifies the smallest possible initial delay, which is one frame.

More general ways to improve throughput rates include simplifying the model, and running the simulation on a faster PC processor. See the Simulink documentation and “Delay and Latency” on page 3-85 for other ideas on improving simulation performance.

Dialog Box



Queue duration (seconds)

The length of signal (in seconds) to buffer to the hardware at the start of the simulation.

Initial output delay (seconds)

The amount of time by which to delay the initial output to the audio device. A value of `0` specifies the smallest possible initial delay, a single frame.

Use default audio device

Directs audio output to the system's default audio device when selected. Deselect to enable the **Audio device ID** parameter and manually enter a device ID number.

Audio device ID

The number of the audio device to receive the audio output. In a system with several audio devices installed, a value of `1` selects the first audio card,

a value of 2 selects the second audio card, and so on. Select **Use default audio device** if the system has only a single audio card installed.

See Also

From Wave Device	DSP Blockset
To Wave File	DSP Blockset
sound	MATLAB

See “Exporting and Playing WAV Files” on page 3-79 for related information.

To Wave File

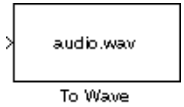
Purpose

Write audio data to file in the Microsoft Wave (`.wav`) format (Windows only).

Library

DSP Sinks

Description



The To Wave File block writes audio data to a Microsoft Wave (`.wav`) file in the uncompressed PCM (pulse code modulation) format. For compatibility reasons, the sample rate of the discrete-time input signal should typically be one of the standard Windows audio device rates (8000, 11025, 22050, or 44100 Hz), although the block supports arbitrary rates.

The input to the block, *u*, can contain audio data from a mono or stereo signal. A mono signal is represented as either a sample-based scalar or frame-based length-M vector, while a stereo signal is represented as a sample-based length-2 vector or frame-based M-by-2 matrix. The amplitude of the input should be in the range ±1. Values outside this range are clipped to the nearest allowable value.

```
wavwrite(u,Fs,bits,'filename')          % Equivalent MATLAB code
```

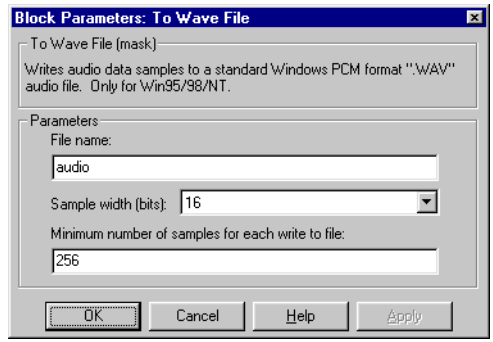
The **Sample Width (bits)** parameter specifies the number of bits used to represent the signal samples in the file. Two settings are available:

- **8** – allocates 8 bits to each sample, allowing a resolution of 256 levels
- **16** – allocates 16 bits to each sample, allowing a resolution of 65536 levels

The 16-bit sample width setting requires more memory but yields better fidelity for double-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 ($L \geq M$). For $L < M$, the block instead writes M consecutive samples during each access. Larger values of L result in fewer file accesses, which reduces run-time overhead.

Dialog Box



File name

The path and name of the file to write. Paths can be relative or absolute.

Sample width (bits)

The number of bits used to represent each signal sample.

Minimum number of samples for each write to file

The number of consecutive samples to write with each file access, L.

See Also

From Wave File	DSP Blockset
To Wave Device	DSP Blockset
To Workspace	Simulink
`wavwrite`	MATLAB

See “Exporting and Playing WAV Files” on page 3-79 for related information.

Transpose

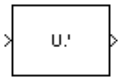
Purpose

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
```

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \end{bmatrix} \xrightarrow{\text{u'}} \begin{bmatrix} u_{11}^* & u_{21}^* \\ u_{12}^* & u_{22}^* \\ u_{13}^* & u_{23}^* \end{bmatrix}$$

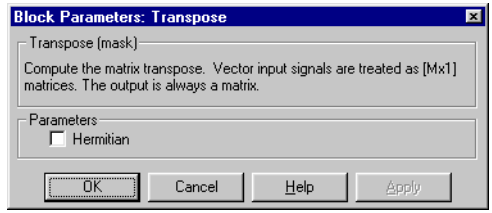
When the **Hermitian** check box is not selected, the block performs the nonconjugate transpose

```
y = u.'         % Equivalent MATLAB code
```

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \end{bmatrix} \xrightarrow{\text{u.'}} \begin{bmatrix} u_{11} & u_{21} \\ u_{12} & u_{22} \\ u_{13} & u_{23} \end{bmatrix}$$

A length-M 1-D vector input is treated as an M-by-1 matrix. The output is always sample-based.

Dialog Box



Hermitian

When selected, specifies the complex conjugate transpose. Tunable, except in Simulink's external mode.

See Also

Permute Matrix	DSP Blockset
Reshape	Simulink
Submatrix	DSP Blockset

Triggered Delay Line

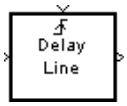
Purpose

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 (in the same simulation step) so that the new output overlaps the previous output by M_o-1 samples. Between triggering events the block ignores input 1 and holds the output at its last value.

The triggering event at input 2 is specified by the **Trigger type** pop-up menu, and can be one of the following:

- **Rising edge** triggers execution of the block when the trigger input rises from a negative value to zero or a positive value, or from zero to a positive value.
- **Falling edge** triggers execution of the block when the trigger input falls from a positive value to zero or a negative value, or from zero to a negative value.
- **Either edge** triggers execution of the block when either a rising or falling edge (as described above) occurs.

The Triggered Delay Line block has zero latency, so the new input appears at the output in the same simulation time step. The output frame period is the same as the input sample period, $T_{fo}=T_{si}$.

Sample-Based Operation

In sample-based operation, the Triggered Delay Line block buffers a sequence of sample-based length-N vector inputs (1-D, row, or column) into a sequence of overlapping sample-based M_o-by-N matrix outputs, where M_o is specified by the **Delay line size** parameter (M_o>1). That is, each input vector becomes a *row* in the sample-based output matrix. When M_o=1, the input is simply passed through to the output, and retains the same dimension. Sample-based full-dimension matrix inputs are not accepted.

Frame-Based Operation

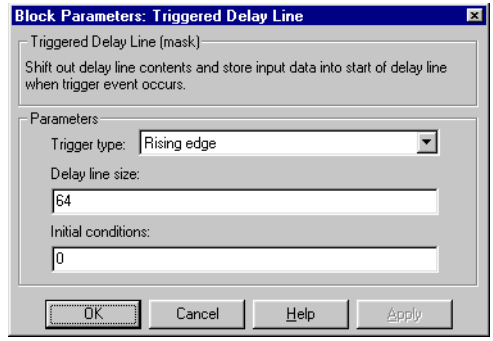
In frame-based operation, the Triggered Delay Line block rebuffers a sequence of frame-based M_i-by-N matrix inputs into an sequence of overlapping

frame-based M_o-by-N matrix outputs, where M_o is the output frame size specified by the **Delay line size** parameter (i.e., the number of consecutive samples from the input frame to rebuffer into the output frame). M_o can be greater or less than the input frame size, M_i. Each of the N input channels is rebuffered independently.

Initial Conditions

The Triggered Delay Line block's buffer is initialized to the value specified by the **Initial condition** parameter. The block always outputs this buffer at the first simulation step (t=0). If the block's output is a vector, the **Initial condition** can be a vector of the same size, or a scalar value to be repeated across all elements of the initial output. If the block's output is a matrix, the **Initial condition** can be a matrix of the same size, a vector (of length equal to the number of matrix rows) to be repeated across all columns of the initial output, or a scalar to be repeated across all elements of the initial output.

Dialog Box



Trigger type

The type of event that triggers the block's execution. Tunable.

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.

Triggered Delay Line

See Also

Buffer	DSP Blockset
Delay Line	DSP Blockset
Unbuffer	DSP Blockset

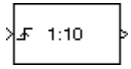
Purpose

Import signal samples from the MATLAB workspace when triggered.

Library

DSP Sources

Description



The Triggered Signal From Workspace block imports signal samples from the MATLAB workspace into the Simulink model when triggered by the control signal at the input port ( ). The **Signal** parameter specifies the name of a MATLAB workspace variable containing the signal to import, or any valid MATLAB expression defining a matrix or 3-D array.

When the **Signal** parameter specifies an M-by-N matrix (M≠1), each of the N columns is treated as a distinct channel. The frame size is specified by the **Samples per frame** parameter, M_o, and the output when triggered is an M_o-by-N matrix containing M_o consecutive samples from each signal channel. For M_o=1, the output is sample-based; otherwise the output is frame-based. For convenience, an imported row vector (M=1) is treated as a single channel, so the output dimension is M_o-by-1.

When the **Signal** parameter specifies an M-by-N-by-P array, the block generates a single page of the array (an M-by-N matrix) at each trigger time. The **Samples per frame** parameter must be set to 1, and the output is always sample-based.

Trigger Event

The triggering event at the input port is specified by the **Trigger type** pop-up menu, and can be one of the following:

- **Rising edge** triggers execution of the block when the trigger input rises from a negative value to zero or a positive value, or from zero to a positive value.
- **Falling edge** triggers execution of the block when the trigger input falls from a positive value to zero or a negative value, or from zero to a negative value.
- **Either edge** triggers execution of the block when either a rising or falling edge (as described above) occurs.

Initial and Final Conditions

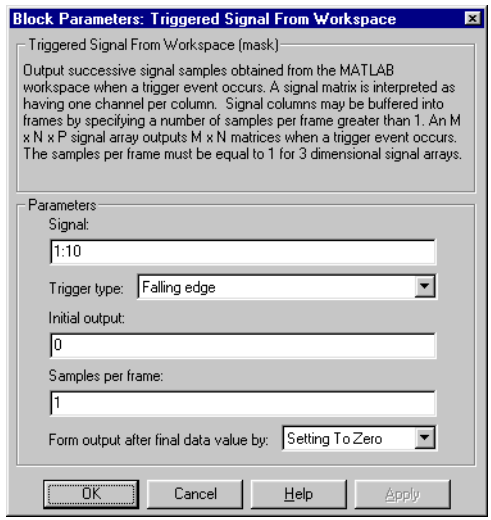
The **Initial output** parameter specifies the output of the block from the start of the simulation until the first trigger event arrives. Between trigger events, the block holds the output value constant at its most recent value (i.e., no linear

interpolation takes place). For single-channel signals, the **Initial output** parameter value can be a vector of length M_o or a scalar to repeat across the M_o elements of the initial output frames. For matrix outputs (M_o-by-N or M-by-N), the **Initial output** parameter value can be a vector of length N to repeat across all rows of the initial outputs, or a scalar to repeat across all elements of the initial matrix outputs.

When the block has output all of the available signal samples, it can start again at the beginning of the signal, or simply repeat the final value or generate zeros until the end of the simulation. (The block does not extrapolate the imported signal beyond the last sample.) The **Form output after final data value by** parameter controls this behavior:

- If **Setting To Zero** is specified, the block generates zero-valued outputs for the duration of the simulation after generating the last frame of the signal.
- If **Holding Final Value** is specified, the block repeats the final sample for the duration of the simulation after generating the last frame of the signal.
- If **Cyclic Repetition** is specified, the block repeats the signal from the beginning after generating the last frame. If there are not enough samples at the end of the signal to fill the final frame, the block zero-pads the final frame as necessary to ensure that the output for each cycle is identical (e.g., the *i*th frame of one cycle contains the same samples as the *i*th frame of any other cycle).

Dialog Box



Signal

The name of the MATLAB workspace variable from which to import the signal, or a valid MATLAB expression specifying the signal.

Trigger type

The type of event that triggers the block's execution. Tunable, except in Simulink's external mode.

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**).

Triggered Signal From Workspace

See Also

From Wave Device	DSP Blockset
From Wave File	DSP Blockset
Sine Wave	DSP Blockset
Signal From Workspace	DSP Blockset
Triggered To Workspace	DSP Blockset

See the sections below for related information:

- “Discrete-Time Signals” on page 3-3
- “Multichannel Signals” on page 3-11
- “Benefits of Frame-Based Processing” on page 3-14
- “Creating Signals Using the Signal From Workspace Block” on page 3-38
- “Importing Signals” on page 3-62

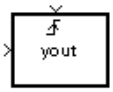
Purpose

Write the input sample to the workspace when triggered.

Library

DSP Sinks

Description



The Triggered Signal To Workspace block creates a matrix or array variable in the workspace, where it stores the acquired inputs at the end of a simulation. An existing variable with the same name is overwritten.

For an M-by-N frame-based input, the block creates an N-column workspace matrix in which each group of M rows represents a single input frame from each of N channels (the most recent frame occupying the last M rows). The maximum size of this workspace variable is limited to P-by-N, where P is the **Maximum number of rows** parameter. (If the simulation progresses long enough for the block to acquire more than P samples, it stores only the most recent P samples.) The **Decimation factor**, D, allows you to store only every Dth input frame.

For an M-by-N sample-based input, the block creates a three-dimensional array in which each M-by-N page represents a single sample from each of M∗N channels (the most recent input matrix occupying the last page). The maximum size of this variable is limited to M-by-N-by-P, where P is the **Maximum number of rows** parameter. (If the simulation progresses long enough for the block to acquire more than P inputs, it stores only the last P inputs.) The **Decimation factor**, D, allows you to store only every Dth input matrix.

The block acquires and buffers a single frame from input 1 whenever it is triggered by the control signal at input 2 (⨍). At all other times, the block ignores input 1. The triggering event at input 2 is specified by the **Trigger type** pop-up menu, and can be one of the following:

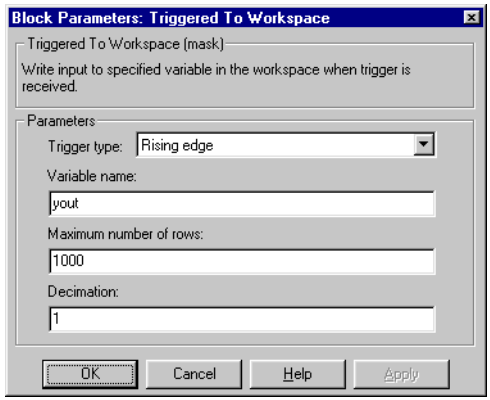
- **Rising edge** triggers execution of the block when the trigger input rises from a negative value to zero or a positive value, or from zero to a positive value.
- **Falling edge** triggers execution of the block when the trigger input falls from a positive value to zero or a negative value, or from zero to a negative value.
- **Either edge** triggers execution of the block when either a rising or falling edge (as described above) occurs.

To save a record of the sample time corresponding to each sample value, check the **Time** box in the **Save to workspace** parameters list of the **Simulation**

Parameters dialog. You can access these parameters by selecting **Parameters** from the **Simulation** menu, and clicking on the **Workspace I/O** tab.

The nontriggered version of this block is To Workspace.

Dialog Box



Trigger type

The type of event that triggers the block's execution. Tunable.

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. The default is `100` rows.

Decimation

The decimation factor, D. The default is `1`.

See Also

Signal From Workspace	DSP Blockset
To Workspace	Simulink

See “Exporting Signals” on page 3-72 for related information.

Purpose

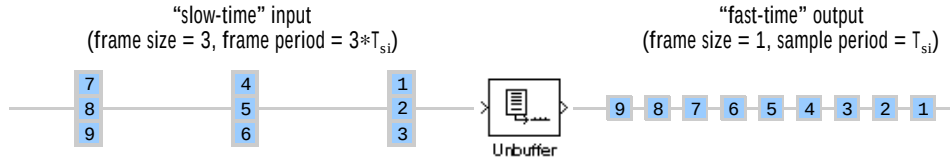
Unbuffer a frame input to a sequence of scalar outputs.

Library

Signal Management / Buffers

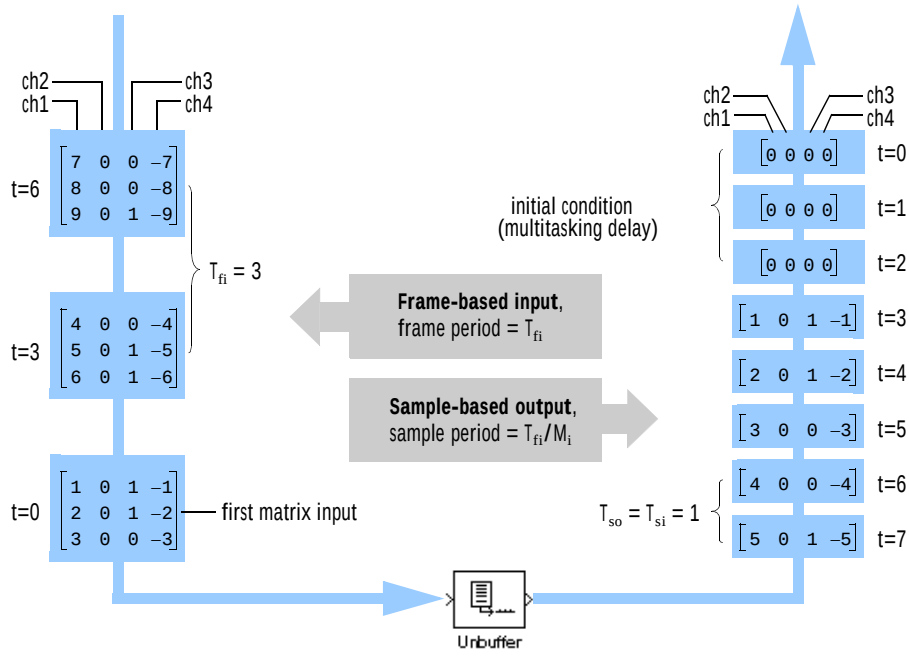
Description

The Unbuffer block unbuffers an M_i-by-N frame-based input into a 1-by-N sample-based output. That is, inputs are unbuffered *row-wise* so that each matrix row becomes an independent time-sample in the output. The rate at which the block receives inputs is generally less than the rate at which the block produces outputs.



The block adjusts the output rate so that the *sample period* is the same at both the input and output, $T_{so}=T_{si}$. Therefore, the output sample period for an input of frame size M_i and frame period T_{fi} is T_{fi}/M_i, which represents a *rate* M_i times higher than the input frame rate. In the example above, the block receives inputs only once every three sample periods, but produces an output once every sample period. To rebuffer frame-based inputs to a larger or smaller frame size, use the Buffer block.

In the model below, the block unbuffers a four-channel frame-based input with frame size 3. The **Initial conditions** parameter is set to zero and the tasking mode is set to multitasking, so the first three outputs are zero vectors (see "Latency" below).



Latency

Zero Latency. The Unbuffer block has *zero tasking latency* in Simulink's single-tasking mode. Zero tasking latency means that the first input sample (received at *t*=0) appears as the first output sample.

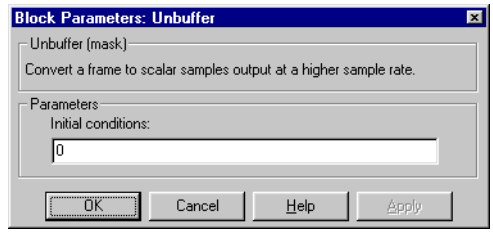
Nonzero Latency. For *multitasking* operation, the Unbuffer block's buffer is initialized with the value specified by the **Initial condition** parameter, and the block begins unbuffering this frame at the start of the simulation. Inputs to the block are therefore delayed by one buffer length, or M_i samples.

The **Initial condition** parameter can be one of the following:

- A scalar to be repeated for the first M_i output samples of every channel
- A length-M_i vector containing the values of the first M_i output samples for every channel
- An M_i-by-N matrix containing the values of the first M_i output samples in each of N channels

See “Excess Algorithmic Delay (Tasking Latency)” on page 3-91 and “The Simulation Parameters Dialog Box” in the Simulink documentation for more information about block rates and Simulink's tasking modes.

Dialog Box



Initial conditions

The value of the block's initial output for cases of nonzero latency; a scalar, vector, or matrix.

See Also

Buffer DSP Blockset

See “Unbuffering a Frame-Based Signal into a Sample-Based Signal” on page 3-60 for related information.

Uniform Decoder

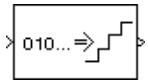
Purpose

Decode an integer input to a floating-point output.

Library

Quantizers

Description



The Uniform Decoder block performs the inverse operation of the Uniform Encoder block, and reconstructs quantized floating-point values from encoded integer input. The block adheres to the definition for uniform decoding specified in *ITU-T Recommendation G.701*.

Inputs can be real or complex values of the following six integer data types: `uint8`, `uint16`, `uint32`, `int8`, `int16`, or `int32`.

The block first casts the integer input values to floating-point values, and then uniquely maps (decodes) them to one of 2^B uniformly spaced floating point values in the range $[-V, (1-2^{1-B})V]$, where B is specified by the **Bits** parameter (as an integer between 2 and 32) and V is a floating-point value specified by the **Peak** parameter. The smallest input value representable by B bits (0 for an unsigned input data type; -2^{B-1} for a signed input data type) is mapped to the value -V. The largest input value representable by B bits (2^B-1 for an unsigned input data type; $2^{B-1}-1$ for a signed input data type) is mapped to the value $(1-2^{1-B})V$. Intermediate input values are linearly mapped to the intermediate values in the range $[-V, (1-2^{1-B})V]$.

To correctly decode values encoded by the Uniform Encoder block, the **Bits** and **Peak** parameters of the Uniform Decoder block should be set to the same values as the **Bits** and **Peak** parameters of the Uniform Encoder block. The **Overflow mode** parameter specifies the Uniform Decoder block's behavior when the integer input is outside the range representable by B bits. If **Saturate** is selected, *unsigned* input values greater than 2^B-1 saturate at 2^B-1; *signed* input values greater than $2^{B-1}-1$ or less than -2^{B-1} saturate at those limits. The real and imaginary components of complex inputs saturate independently.

If **Wrap** is selected, *unsigned* input values, u, greater than 2^B-1 are wrapped back into the range $[0, 2^B-1]$ using mod-2^B arithmetic.

```
u = mod(u,2^B)                % Equivalent MATLAB code
```

Signed input values, u, greater than 2^{B-1}-1 or less than -2^{B-1} are wrapped back into that range using mod-2^B arithmetic.

```
u = (mod(u+2^B/2,2^B)-(2^B/2)) % Equivalent MATLAB code
```

The real and imaginary components of complex inputs wrap independently.

The **Output type** parameter specifies whether the decoded floating-point output is single or double precision. Either level of output precision can be used with any of the six integer input data types.

Example

Consider a Uniform Decoder block with the following parameter settings:

- **Peak** = 2
- **Bits** = 3

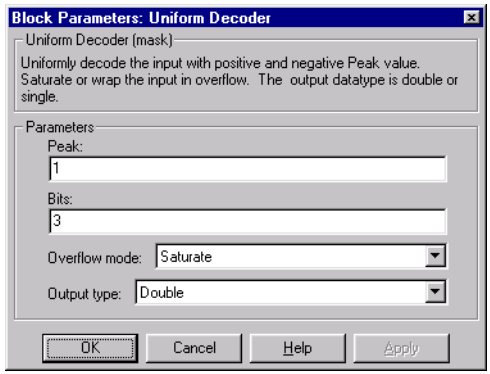
The input to the block is the `uint8` output of a Uniform Encoder block with comparable settings: **Peak** = 2, **Bits** = 3, and **Output type** = **Unsigned**. (Comparable settings ensure that inputs to the Uniform Decoder block do not saturate or wrap. See the example on the Uniform Encoder reference page for more about these settings.)

The real and complex components of each input are independently mapped to one of 2^3 distinct levels in the range `[-2.0,1.5]`.

```
0  is mapped to  -2.0
1  is mapped to  -1.5
2  is mapped to  -1.0
3  is mapped to  -0.5
4  is mapped to   0.0
5  is mapped to   0.5
6  is mapped to   1.0
7  is mapped to   1.5
```

Uniform Decoder

Dialog Box



Peak

The largest amplitude represented in the encoded input. To correctly decode values encoded with the Uniform Encoder block, set the **Peak** parameters in both blocks to the same value.

Bits

The number of input bits, B, used to encode the data. (This can be less than the total number of bits supplied by the input data type.) To correctly decode values encoded with the Uniform Encoder block, set the **Bits** parameters in both blocks to the same value.

Overflow mode

The block's behavior when the integer input is outside the range representable by B bits. Out-of-range inputs can either saturate at the extreme value, or wrap back into range.

Output type

The precision of the floating-point output, `single` or `double`.

References

General Aspects of Digital Transmission Systems: Vocabulary of Digital Transmission and Multiplexing, and Pulse Code Modulation (PCM) Terms, International Telecommunication Union, ITU-T Recommendation G.701, March, 1993

See Also

Data Type Conversion	Simulink
Quantizer	Simulink
Uniform Encoder	DSP Blockset
udecode	Signal Processing Toolbox
uencode	Signal Processing Toolbox

Uniform Encoder

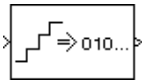
Purpose

Quantize and encode a floating-point input to an integer output.

Library

Quantizers

Description



The Uniform Encoder block performs the following two operations on each floating-point sample in the input vector or matrix:

1. Quantizes the value using the same precision
2. Encodes the quantized floating-point value to an integer value

In the first step, the block quantizes an input value to one of 2^B uniformly spaced levels in the range $[-V, (1-2^{1-B})V]$, where B is specified by the **Bits** parameter and V is specified by the **Peak** parameter. The quantization process rounds both positive and negative inputs *downward* to the nearest quantization level, with the exception of those that fall exactly on a quantization boundary. The real and imaginary components of complex inputs are quantized independently.

The number of bits, B, can be any integer value between 2 and 32, inclusive. Inputs greater than $(1-2^{1-B})V$ or less than -V saturate at those respective values. The real and imaginary components of complex inputs saturate independently.

In the second step, the quantized floating-point value is uniquely mapped (encoded) to one of 2^B integer values. If the **Output type** is set to **Unsigned integer**, the smallest quantized floating-point value, -V, is mapped to the integer 0, and the largest quantized floating-point value, $(1-2^{1-B})V$, is mapped to the integer 2^B-1. Intermediate quantized floating-point values are linearly (uniformly) mapped to the intermediate integers in the range $[0, 2^B-1]$. For efficiency, the block automatically selects an *unsigned* output data type (`uint8`, `uint16`, or `uint32`) with the minimum number of bits equal to or greater than B.

If the **Output type** is set to **Signed integer**, the smallest quantized floating-point value, -V, is mapped to the integer -2^{B-1}, and the largest quantized floating-point value, $(1-2^{1-B})V$, is mapped to the integer $2^{B-1}-1$. Intermediate quantized floating-point values are linearly mapped to the intermediate integers in the range $[-2^{B-1}, 2^{B-1}-1]$. The block automatically selects a *signed* output data type (`int8`, `int16`, or `int32`) with the minimum number of bits equal to or greater than B.

Inputs can be real or complex, double or single precision. The output data types that the block uses are shown in the table below. Note that most of the blocks in the DSP Blockset accept only double precision inputs. Use the Simulink Data Type Conversion block to convert integer data types to double precision. See “Working with Data Types” in the Simulink documentation for a complete discussion of data types, as well as a list of Simulink blocks capable of reduced-precision operations.

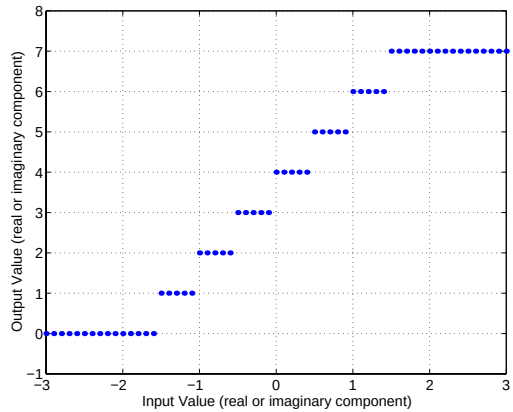
Bits	Unsigned Integer	Signed Integer
2 to 8	`uint8`	`int8`
9 to 16	`uint16`	`int16`
17 to 32	`uint32`	`int32`

The Uniform Encoder block operations adhere to the definition for uniform encoding specified in *ITU-T Recommendation G.701*.

Example

The figure below illustrates uniform encoding with the following parameter settings:

- **Peak** = 2
- **Bits** = 3
- **Output type** = **Unsigned**



The real and complex components of each input (horizontal axis) are independently quantized to one of 2^3 distinct levels in the range `[-2,1.5]` and then mapped to one of 2^3 integer values in the range `[0,7]`.

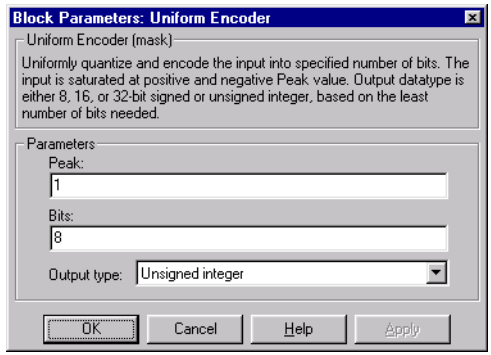
```
-2.0  is mapped to  0
-1.5  is mapped to  1
-1.0  is mapped to  2
-0.5  is mapped to  3
 0.0  is mapped to  4
 0.5  is mapped to  5
 1.0  is mapped to  6
 1.5  is mapped to  7
```

The table below shows the results for a few particular inputs.

Input	Quantized Input	Output	Notes
`1.6`	`1.5+0.0i`	`7+4i`	
`-0.4`	`-0.5+0.0i`	`3+4i`	
`-3.2`	`-2.0+0.0i`	`4i`	Saturation (real)
`0.4-1.2i`	`0.0-1.5i`	`4+i`	
`0.4-6.0i`	`0.0-2.0i`	`4`	Saturation (imaginary)
`-4.2+3.5i`	`-2.0+2.0i`	`7i`	Saturation (real and imag)

The output data type is automatically set to `uint8`, the most efficient format for this input range.

Dialog Box



Peak

The largest input amplitude to be encoded, V. Real or imaginary input values greater than $(1-2^{1-B})V$ or less than -V saturate (independently for complex inputs) at those limits.

Bits

The number of levels at which to quantize the floating-point input. (Also the number of bits needed to represent the integer output.)

Output type

The data type of the block's output, **Unsigned integer** or **Signed integer**. Unsigned outputs are `uint8`, `uint16`, or `uint32`, while signed outputs are `int8`, `int16`, or `int32`.

References

General Aspects of Digital Transmission Systems: Vocabulary of Digital Transmission and Multiplexing, and Pulse Code Modulation (PCM) Terms, International Telecommunication Union, ITU-T Recommendation G.701, March, 1993

See Also

Data Type Conversion	Simulink
Quantizer	Simulink
Uniform Decoder	DSP Blockset
udecode	Signal Processing Toolbox
uencode	Signal Processing Toolbox

Unwrap

Purpose

Unwrap an input containing radian phase angles.

Library

Signal Operations

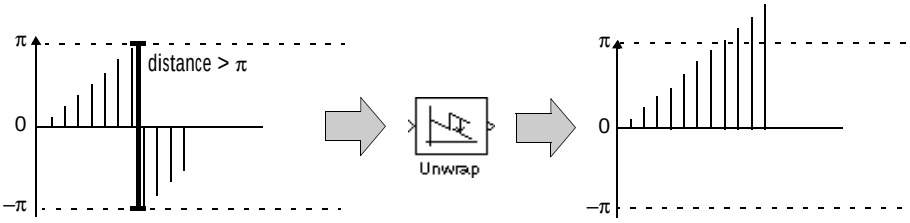
Description

The Unwrap block removes phase discontinuities in each column of the M-by-N input matrix, *u*, by adding or subtracting an appropriate multiple of 2π to each element. The block recognizes a phase discontinuity, or *phase jump*, whenever a given input element, $u_{i,j}$, differs from the preceding element in the same column by an absolute amount greater than the absolute value of the specified **Tolerance** parameter, α.

To eliminate the jumps in the input, the block scans down each column and adds $2\pi k$ to all elements following a jump (including the discontinuous element itself). The value of *k* is initialized to 0, and is then incremented by 1 for each successive negative jump in the column ($|u_{i,j} - u_{i-1,j}| > |\alpha|$; $u_{i,j} < u_{i-1,j}$), and decremented by 1 for each successive positive jump in the column ($|u_{i,j} - u_{i-1,j}| > |\alpha|$; $u_{i,j} > u_{i-1,j}$).

The value of *k* is reset to 0 for each successive sample-based input. For frame-based inputs, the value of *k* is retained across successive inputs so that unwrapping can proceed continuously across the frame breaks.

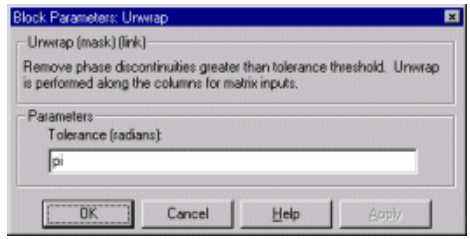
The figure below illustrates a single channel being unwrapped with the default **Tolerance** setting of π. The block recognizes a negative jump of 2π when the ascending sequence of radian phase angles rolls around from π to -π. Beginning with the -π element, the block adds 2π to all subsequent values in the column. If another negative jump is detected in the same column, the block adds 4π to all subsequent values, and so on.



The output has the same dimension and frame status as the input. 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, and the output retains the dimensions of the input.

Dialog Box



Tolerance

The jump size that is recognized 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. Tunable, except in Simulink's external mode.

See Also

`unwrap` MATLAB

Upsample

Purpose

Resample an input at a higher rate by inserting zeros.

Library

Signal Operations

Description

The Upsample block resamples each channel of the M_i-by-N input at a rate L times higher than the input sample rate by inserting L-1 zeros between consecutive samples. The integer L is specified by the **Upsample factor** parameter. The **Sample offset** parameter delays the output samples by an integer number of sample periods D, where $0 \le D < L$, so that any of the L possible output phases can be selected.

Sample-Based Operation

When the input is sample-based, the block treats each of the M∗N matrix elements as an independent channel, and upsamples each channel over time. The **Frame-based mode** parameter must be set to **Maintain input frame size**. The output sample rate is L times higher than the input sample rate ($T_{so} = T_{si}/L$), and the input and output sizes are identical.

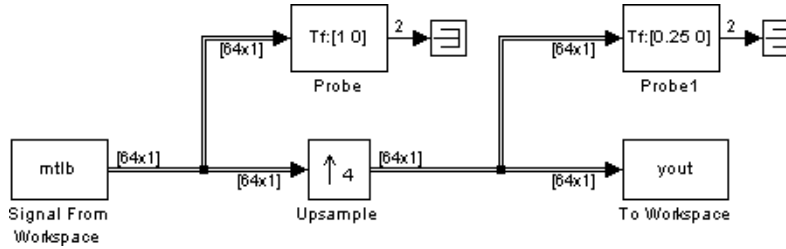
Frame-Based Operation

When the input is frame-based, the block treats each of the N input columns as a frame containing M_i sequential time samples from an independent channel. The block upsamples each channel independently by inserting L-1 rows of zeros between each row in the input matrix. The **Frame-based mode** parameter determines how the block adjusts the rate at the output to accommodate the added rows. There are two available options:

- **Maintain input frame size**

  The block generates the output at the faster (upsampled) rate by using a proportionally shorter frame *period* at the output port than at the input port. For upsampling by a factor of L, the output frame period is L times shorter than the input frame period ($T_{fo} = T_{fi}/L$), but the input and output frame sizes are equal.

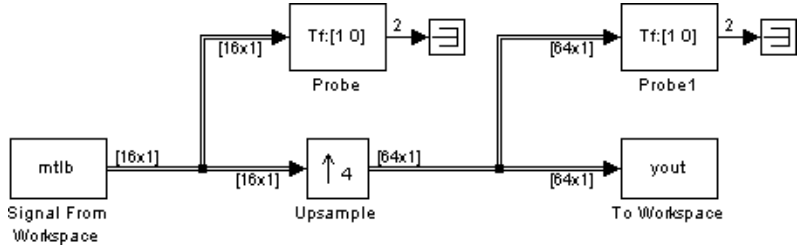
  The model below shows a single-channel input with a frame period of 1 second being upsampled by a factor of 4 to a frame period of 0.25 seconds. The input and output frame sizes are identical.



- **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 Simulink's 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.

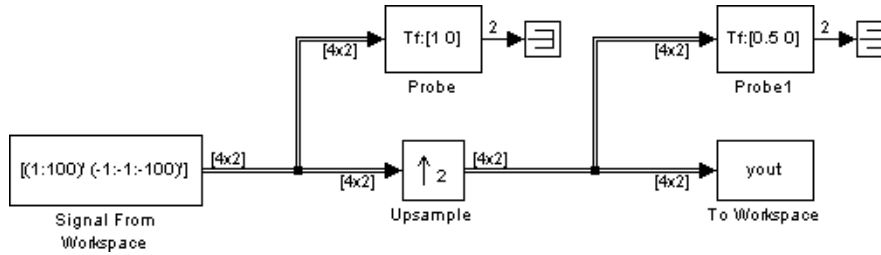
Nonzero Latency. The Upsample block has tasking latency only for multirate operation in Simulink's multitasking mode:

- In sample-based mode, the initial condition for each channel appears as output sample D+1, and is followed by L-1 inserted zeros. The channel's first input appears as output sample D+L+1. The **Initial condition** value can be an M_i-by-N matrix containing one value for each channel, or a scalar to be applied to all signal channels.
- In frame-based mode, the first row of the initial condition matrix appears as output sample D+1, and is followed by L-1 inserted rows of zeros, the second row of the initial condition matrix, and so on. The first row of the first input matrix appears in the output as sample M_iL+D+1. The **Initial condition** value can be an M_i-by-N matrix, or a scalar to be repeated across all elements of the M_i-by-N matrix. See the example below for an illustration of this case.

See “Excess Algorithmic Delay (Tasking Latency)” on page 3-91 and “The Simulation Parameters Dialog Box” in the Simulink documentation for more information about block rates and Simulink's 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 deselecting the **Probe width** and **Probe complex signal** check boxes (if desired).

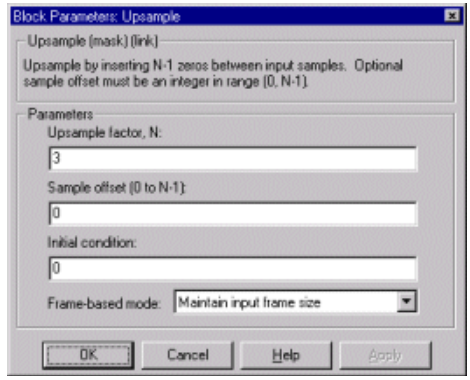
This model is multirate because there are at least two distinct frame rates, as shown by the two Probe blocks. To run this model in Simulink's multitasking mode, select **Fixed-step** and **discrete** from the **Type** controls in the **Solver** panel of the **Simulation Parameters** dialog box, and select **MultiTasking** from the **Mode** parameter. Also set the **Stop time** to `30`.

Run the model and look at the output, `yout`. The first few samples of each channel are shown below.

```
yout =

     0     0
    11   -11
     0     0
    12   -12
     0     0
    13   -13
     0     0
    14   -14
     0     0
     1    -1
     0     0
     2    -2
     0     0
     3    -3
     0     0
     4    -4
     0     0
     5    -5
     0     0
```

Since we ran this frame-based multirate model in multitasking mode, the first row of the initial condition matrix appears as output sample 2 (i.e., sample D+1, where D is the **Sample offset** value). It is followed by the other three initial condition rows, each separated by L-1 inserted rows of zeros, where L is the **Upsample factor** value of 2. The first row of the first input matrix appears in the output as sample 10 (i.e., sample M_iL+D+1, where M_i is the input frame size).

Dialog Box



Upsample factor

The integer factor, L, by which to increase the input sample rate.

Sample offset

The sample offset, D, which must be an integer in the range [0,L-1].

Initial condition

The value with which the block is initialized for cases of nonzero latency, a scalar or matrix. This value (first row in frame-based mode) appears in the output as sample D+1.

Frame-based mode

For frame-based operation, the method by which to implement the upsampling: **Maintain input frame size** (i.e., increase the frame rate), or **Maintain input frame rate** (i.e., increase the frame size). The **Framing** parameter must be set to **Maintain input frame size** for sample-base inputs.

See Also

Downsample	DSP Blockset
FIR Interpolation	DSP Blockset
FIR Rate Conversion	DSP Blockset
Repeat	DSP Blockset

Variable Fractional Delay

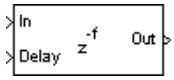
Purpose

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*, is an M_i-by-N matrix of floating-point values in the range $0 \le v \le D$ that specifies the number of sample intervals to delay each channel of the input.

A 1-D vector input is treated as an M_i-by-1 matrix, and the output is 1-D.

The **Initial conditions** parameter specifies the values in the block's memory at the start of the simulation in the same manner as for the Variable Integer Delay block. See the section on sample-based initial conditions there for complete information.

Frame-Based Operation

When the input is frame-based, the block treats each of the N input columns as a frame containing M_i sequential time samples from an independent channel.

The input to the `Delay` port, *v*, contains floating-point values in the range $0 \le v \le D$ specifying the number of sample intervals to delay the current input. The input to the `Delay` port can be:

- An M_i-by-N matrix containing the number of sample intervals to delay *each* sample in *each* channel of the current input
- An M_i-by-1 matrix containing the number of sample intervals to delay each sample in *every* channel of the current input
- A 1-by-N matrix containing the number of sample intervals to delay *every* sample in each channel of the current input

For example, if *v* is the M_i-by-1 matrix `[v(1) v(2) ... v(Mi)]'`, the earliest sample in the current frame is delayed by `v(1)` fractional sample intervals, the following sample in the frame is delayed by `v(2)` fractional sample intervals, and so on. The set of fractional delays contained in *v* is applied identically to every channel of a multichannel input.

The **Initial conditions** parameter specifies the values in the block's memory at the start of the simulation in the same manner as for the Variable Integer Delay block. See the section on frame-based initial conditions there for complete information.

Interpolation Modes

The delay value specified at the `Delay` port is used as an index into the block's memory, `U`, which stores the D+1 most recent samples received at the `In` port for each channel. For example, an integer delay of `5` on a scalar input sequence retrieves and outputs the fifth most recent input sample from the block's memory, `U(6)`. Fractional delays are computed by interpolating between stored samples; the two available interpolation modes are described below.

Linear Interpolation Mode. For noninteger delays, at each sample time the **Linear Interpolation** mode uses the two samples in memory nearest to the specified delay to compute a value for the sample at that time. If `v` is the specified fractional delay for a scalar input, the output sample, `y`, is computed as follows.

```
vi = floor(v)                  % vi = integer delay
vf = v-vi                      % vf = fractional delay
y = (1-vf)*U(vi) + vf*U(vi+1)
```

Delay values less than 0 are clipped to 0, and delay values greater than D are clipped to D, where D is the **Maximum delay**. Note that a delay value of 0 causes the block to pass through the current input sample, `U(1)`, in the same simulation step that it is received.

FIR Interpolation Mode. In **FIR Interpolation** mode, the block computes a value for the sample at the desired delay by applying an FIR filter of order 2P to the stored samples on either side of the desired delay, where P is the **Interpolation filter half-length**. For periodic signals, a larger value of P (i.e., a higher order filter) yields a better estimate of the sample at the specified delay. A value between 4 and 6 for this parameter (i.e. a 7th to 11th order filter) is usually adequate.

A vector of 2P filter tap weights is precomputed at the start of the simulation for each of Q-1 discrete points between input samples, where Q is specified by the **Interpolation points per input sample** parameter. For a delay corresponding to one of the Q interpolation points, the unique filter computed for that interpolation point is applied to obtain a value for the sample at the specified delay. For delay times that fall between interpolation points, the value computed at the nearest interpolation point is used. Since Q controls the number of locations where a unique interpolation filter is designed, a larger value results in a better estimate of the sample at a given delay.

Note that increasing the **Interpolation filter half length** (P) increases the number of computations performed per input sample, as well as the amount of memory needed to store the filter coefficients. Increasing the **Interpolation points per input sample** (Q) increases the simulation's memory requirements but does not affect the computational load per sample.

The **Normalized input bandwidth** parameter allows you to take advantage of the bandlimited frequency content of the input. For example, if you know that the input signal does not have frequency content above $F_s/4$, you can specify a value of `0.5` for the **Normalized input bandwidth** to constrain the frequency content of the output to that range.

(Each of the Q interpolation filters can be considered to correspond to one output phase of an "upsample-by-Q" FIR filter. In this view, the **Normalized input bandwidth** value is used to improve the stopband in critical regions, and to relax the stopband requirements in frequency regions where there is no signal energy.)

For delay values less than P/2-1, the output is computed using linear interpolation. Delay values greater than D are clipped to D, where D is the **Maximum delay**.

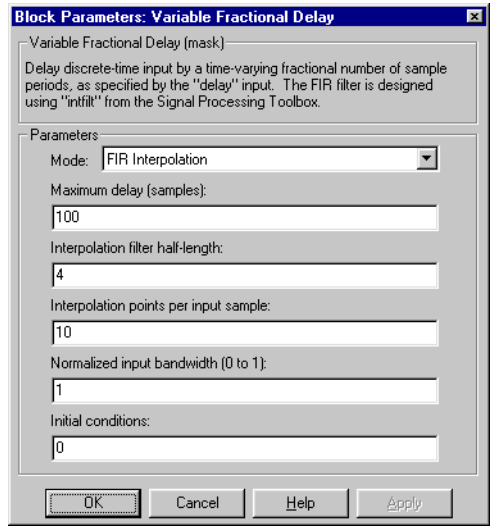
The block uses the `intfilt` function in the Signal Processing Toolbox to compute the FIR filters.

Note When the Variable Fractional Delay block is used in a feedback loop, at least one block with nonzero delay (e.g., an Integer Delay block with **Delay** > `0`) should be included in the loop as well. This prevents the occurrence of an algebraic loop if the delay of the Variable Fractional Delay block is driven to zero.

Examples

The `dspafxf` demo illustrates an audio flanger system built around the Variable Fractional Delay block.

Dialog Box



Mode

The method by which to interpolate between adjacent stored samples to obtain a value for the sample indexed by the input at the `Delay` port.

Maximum delay

The maximum delay that the block can produce, D. Delay input values exceeding this maximum are clipped at the maximum.

Interpolation filter half-length

Half the number of input samples to use in the FIR interpolation filter.

Interpolation points per input sample

The number of points per input sample, Q, at which a unique FIR interpolation filter is computed.

Normalized input bandwidth

The bandwidth to which the interpolated output samples should be constrained. A value of `1` specifies half the sample frequency.

Initial conditions

The values with which the block's memory is initialized. See the Variable Integer Delay block for more information.

See Also

Integer Delay	DSP Blockset
Unit Delay	Simulink
Variable Integer Delay	DSP Blockset

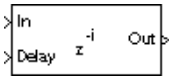
Purpose

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 Integer Delay block in the following ways.

Variable Integer Delay	Integer Delay
Delay is provided as an input to the `Delay` port.	Delay is specified as a parameter setting in the dialog box.
Delay can vary with time; for example, for a frame-based input, the *n*th element's delay in the first input frame can differ from the *n*th element's delay in the second input frame.	Delay cannot vary with time; for example, for a frame-based input, the *n*th element's delay is the same for every input frame.

Sample-Based Operation

When the input is an M-by-N sample-based matrix, the block treats each of the M∗N matrix elements as an independent channel, and applies the delay at the `Delay` port to each channel.

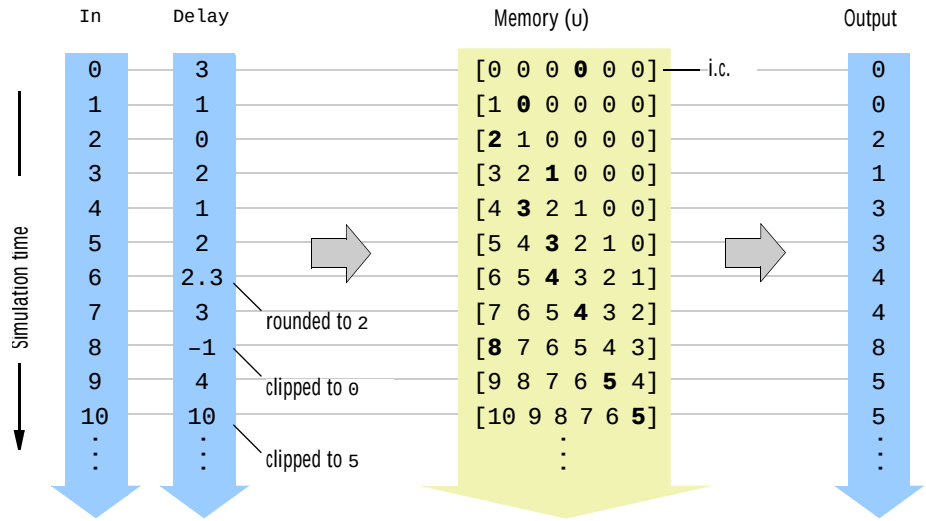
The Variable Integer Delay block stores the D+1 most recent samples received at the `In` port for each channel. At each sample time the block outputs the stored sample(s) indexed by the input to the `Delay` port.

For example, if the input to the `In` port, `u`, is a scalar signal, the block stores a vector, `U`, of the D+1 most recent signal samples. If the current input sample is `U(1)`, the previous input sample is `U(2)`, and so on, then the block's output is

```
y = U(v+1);          % Equivalent MATLAB code
```

where `v` is the input to the `Delay` port. Note that a delay value of 0 (`v=0`) causes the block to pass through the sample at the `In` port in the same simulation step that it is received. The block's memory is initialized to the **Initial conditions** value at the start of the simulation (see below).

The figure below shows the block output for a scalar ramp sequence at the `In` port, a **Maximum delay** of 5, an **Initial conditions** of 0, and a variety of different delays at the `Delay` port.



Note that the current input at each time-step is immediately stored in memory as `U(1)`. This allows the current input to be available at the output for a delay of 0 (`v=0`).

The **Initial conditions** parameter specifies the values in the block's memory at the start of the simulation. Unlike the Integer Delay block, the Variable

Integer Delay block does not have a fixed *initial delay* period during which the initial conditions appear at the output. Instead, the initial conditions are propagated to the output only when they are indexed in memory by the value at the `Delay` port. Both fixed and time-varying initial conditions can be specified in a variety of ways to suit the dimensions of the input sequence.

Fixed Initial Conditions. The settings shown below specify *fixed* initial conditions. For a fixed initial condition, the block initializes each of D samples in memory to the value entered in the **Initial conditions** parameter. A fixed initial condition in sample-based mode can be specified in one of the following ways:

- *Scalar* value with which to initialize every sample of every channel in memory. For a general M-by-N input and the parameter settings below,

  the block initializes 100 M-by-N matrices in memory with zeros.

- *Array* of size M-by-N-by-D. In this case, you can specify different fixed initial conditions for each channel. See the *Array* bullet in "Time-Varying Initial Conditions" below for details.

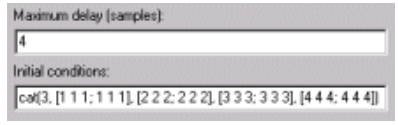
Initial conditions cannot be specified by full matrices.

Time-Varying Initial Conditions. The following settings specify *time-varying* initial conditions. For a time-varying initial condition, the block initializes each of D samples in memory to one of the values entered in the **Initial conditions** parameter. This allows you to specify a unique output value for each sample in memory. A time-varying initial condition in sample-based mode can be specified in one of the following ways:

- *Vector* containing D elements with which to initialize memory samples `U(2:D+1)`, where D is the **Maximum delay**. For a scalar input and the parameters shown below,

the block initializes `U(2:6)` with values `[-1, -1, -1, 0, 1]`.

- *Array* of dimension M-by-N-by-D with which to initialize memory samples `U(2:D+1)`, where D is the **Maximum delay** and M and N are the number of rows and columns, respectively, in the input matrix. For a 2-by-3 input and the parameters below,



the block initializes memory locations `U(2:5)` with values

$$U(2) = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}, U(3) = \begin{bmatrix} 2 & 2 & 2 \\ 2 & 2 & 2 \end{bmatrix}, U(4) = \begin{bmatrix} 3 & 3 & 3 \\ 3 & 3 & 3 \end{bmatrix}, U(5) = \begin{bmatrix} 4 & 4 & 4 \\ 4 & 4 & 4 \end{bmatrix}$$

An array initial condition can only be used with matrix inputs.

Initial conditions cannot be specified by full matrices.

Frame-Based Operation

When the input is an M-by-N frame-based matrix, the block treats each of the N input columns as a frame containing M sequential time samples from an independent channel.

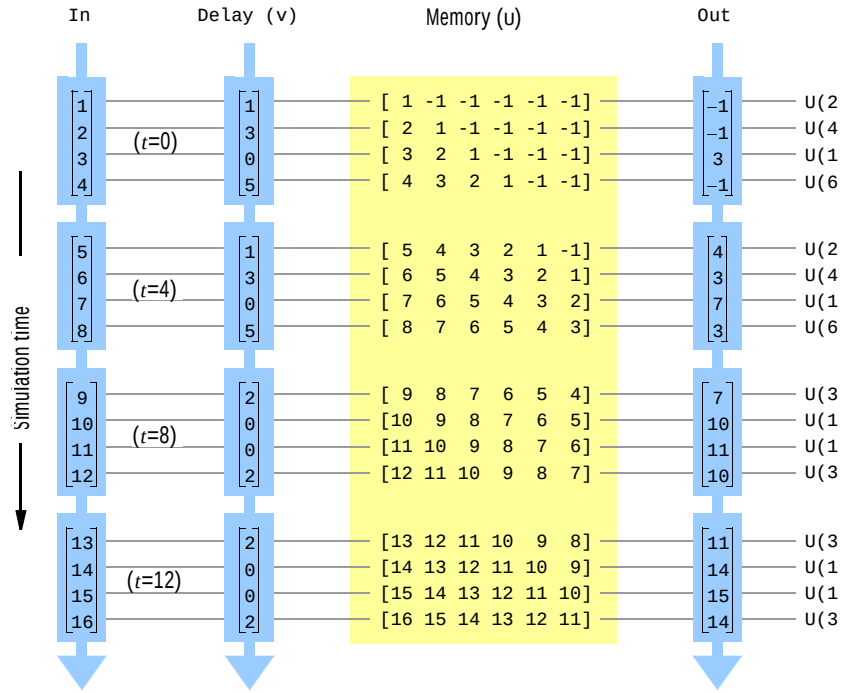
In frame-based mode, the input at the `Delay` port can be a scalar value to uniformly delay every sample in every channel. It can also be a length-M vector, `v = [v(1) v(2) ... v(M)]`, containing one delay for each sample in the input frame(s). The set of delays contained in vector `v` is applied identically to every channel of a multichannel input. The `Delay` port entry can also be a length-N vector, containing one delay for each channel.

Vector `v` *does not* specify when the samples in the current input frame will appear in the output. Rather, `v` indicates which *previous* input samples (stored in memory) should be included in the current output frame. The first sample in the current output frame is the input sample `v(1)` intervals earlier in the

sequence, the second sample in the current output frame is the input sample v(2) intervals earlier in the sequence, and so on.

The illustration below shows how this works for an input with a sample period of 1 and frame size of 4. The **Maximum delay** (Dmax) is 5, and the **Initial conditions** parameter is set to -1. The delay input changes from [1 3 0 5] to [2 0 0 2] after the second input frame. Note that the samples in each output frame are the values in memory indexed by the elements of v.

```
y(1) = U(v(1)+1)
y(2) = U(v(2)+1)
y(3) = U(v(3)+1)
y(4) = U(v(4)+1)
```



The **Initial conditions** parameter specifies the values in the block's memory at the start of the simulation. Both fixed and time-varying initial conditions can be specified.

Fixed Initial Conditions. The settings shown below specify *fixed* initial conditions. For a fixed initial condition, the block initializes each of D samples in memory to the value entered in the **Initial conditions** parameter. A fixed initial condition in frame-based mode can be one of the following:

- *Scalar* value with which to initialize every sample of every channel in memory. For a general M-by-N input with the parameter settings below,

  the block initializes five samples in memory with zeros.

- *Array* of size 1-by-N-by-D. In this case, you can specify different fixed initial conditions for each channel. See the *Array* bullet in "Time-Varying Initial Conditions" below for details.

Initial conditions cannot be specified by full matrices.

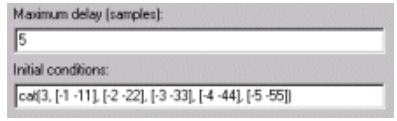
Time-Varying Initial Conditions. The following setting specifies a *time-varying* initial condition. For a time-varying initial condition, the block initializes each of D samples in memory to one of the values entered in the **Initial conditions** parameter. This allows you to specify a unique output value for each sample in memory. A time-varying initial condition in frame-based mode can be specified in the following way:

- *Vector* of dimensions 1-by-D. In this case, all channels have the same set of time-varying initial conditions specified by the entries of the vector. For the ramp input `[100; 100]'` with a frame size of 4, delay of 5, and the parameter settings below,

  the block outputs the following sequence of frames at the start of the simulation.

$$\begin{bmatrix} -1 & -1 \\ -2 & -2 \\ -3 & -3 \\ -4 & -4 \end{bmatrix}, \begin{bmatrix} -5 & -5 \\ 1 & 1 \\ 2 & 2 \\ 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 \\ 5 & 5 \\ 6 & 6 \\ 7 & 7 \end{bmatrix}, \ldots$$

- *Array* of size 1-by-N-by-D. In this case, you can specify different time-varying initial conditions for each channel. For the ramp input `[100; 100]'` with a frame size of 4, delay of 5, and the parameter settings below,



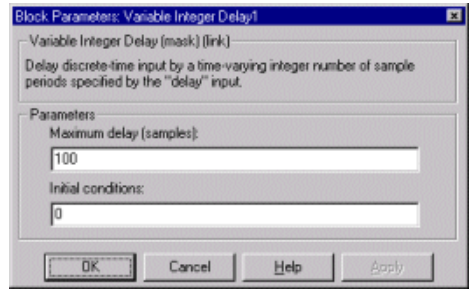
  the block outputs the following sequence of frames at the start of the simulation.

$$\begin{bmatrix} -1 & -11 \\ -2 & -22 \\ -3 & -33 \\ -4 & -44 \end{bmatrix}, \begin{bmatrix} -5 & -55 \\ 1 & 1 \\ 2 & 2 \\ 3 & 3 \end{bmatrix}, \begin{bmatrix} 4 & 4 \\ 5 & 5 \\ 6 & 6 \\ 7 & 7 \end{bmatrix}, \ldots$$

  Note that by specifying a 1-by-N-by-D initial condition array such that each 1-by-N vector entry is identical, you can implement different *fixed* initial conditions for each channel.

Initial conditions cannot be specified by full matrices.

Dialog Box



Maximum delay

The maximum delay that the block can produce for any sample. Delay input values exceeding this maximum are clipped at the maximum.

Initial conditions

The values with which the block's memory is initialized.

See Also

Integer Delay	DSP Blockset
Variable Fractional Delay	DSP Blockset

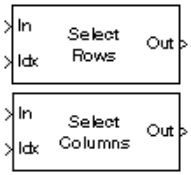
Purpose

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.

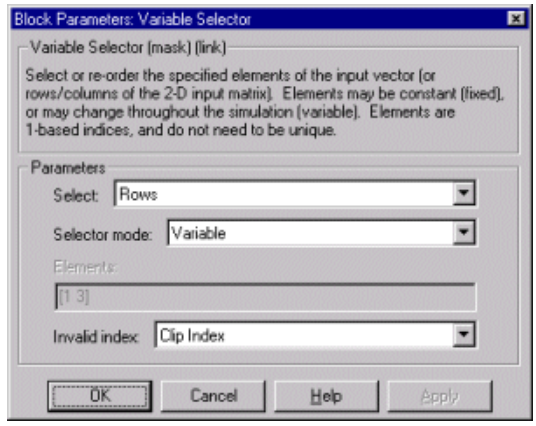
When the input is a 1-D vector, the **Select** parameter is ignored; the output is a 1-D vector of length L containing those elements specified by the length-L indexing vector.

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



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. Tunable.

Invalid index

Response to an invalid index value. Tunable.

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:

$$y_j = \sigma_j^2 = \frac{\sum_{i=1}^{M} |u_{ij} - \mu_j|^2}{M-1} \qquad 1 \le j \le N$$

where μ_j is the mean of the *j*th column. For complex inputs, the output is the *total variance* for each column in u, which is the sum of the real and imaginary variances for that column:

$$\sigma_j^2 = \sigma_{j,Re}^2 + \sigma_{j,Im}^2$$

The frame status of the output is the same as that of the input.

Running Operation

When the **Running variance** check box is selected, the block tracks the variance of each channel in a *time-sequence* of M-by-N inputs. For sample-based inputs, the output is a sample-based M-by-N matrix with each

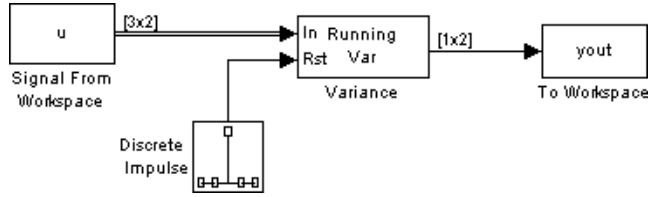
element y_{ij} containing the variance of element u_{ij} over all inputs since the last reset. For frame-based inputs, the output is a frame-based M-by-N matrix with each element y_{ij} containing the variance of the jth column over all inputs since the last reset, up to and including element u_{ij} of the current input.

If the **Reset port** parameter is set to **Non-zero sample**, the optional `Rst` port is enabled and the block resets the running variance when the scalar input at the `Rst` port is nonzero. (The `Rst` port can be disabled by setting the **Reset port** parameter to **None**.)

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.

Example

The Variance block in the model below calculates the running variance of a frame-based 3-by-2 (two-channel) matrix input, `u`. The running variance is reset at t=2 by an impulse to the block's `Rst` port.



The Variance block has the following settings:

- **Running variance** = ☑
- **Reset port** = **Non-zero sample**

The Signal From Workspace block has the following settings:

- **Signal** = `u`
- **Sample time** = `1/3`
- **Samples per frame** = `3`

where

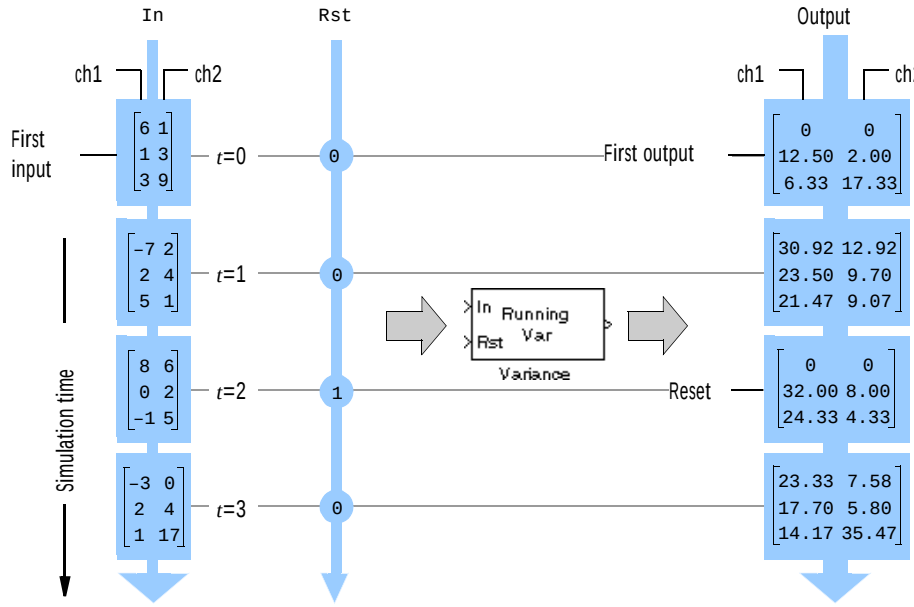
```
u = [6 1 3 -7 2 5 8 0 -1 -3 2 1;1 3 9 2 4 1 6 2 5 0 4 17]'
```

The Discrete Impulse block has the following settings:

- **Delay (samples)** = `2`

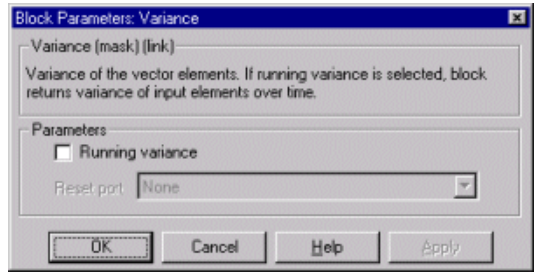
- **Sample time** = 1
- **Samples per frame** = 1

The block's operation is shown in the figure below.



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

Dialog Box



Running variance

Enables running operation when selected.

Reset port

Enables the `Rst` input port when set to Non-zero sample, and disables the `Rst` input port when set to None.

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 tool, similar to a digital oscilloscope, for displaying time-domain, frequency-domain, or user-defined signals. The scope window, display-property settings, axis-property settings, and line-property settings are shared with the Spectrum Scope block.

The input to this block can be any M-by-N matrix or 1-D vector, where 1-D vectors are treated as column vectors. The frame-status for inputs are ignored; the input to the block is always assumed to be a data frame, even if the input is not identified as a frame. Thus, any M-by-N matrix input is interpreted as having N independent channels of data, each with M consecutive samples to be plotted sequentially across the horizontal axis of the plot.

The Vector Scope is most commonly used to plot consecutive time samples (from a frame-based vector). However, it is just as appropriate to use the Vector Scope to plot vectors containing data such as filter coefficients or spectral magnitudes.

Displaying Data

The domain of the data is specified by the **Input domain** parameter under the **Scope properties** check box, and can be **Time**, **Frequency**, or **User-defined**.

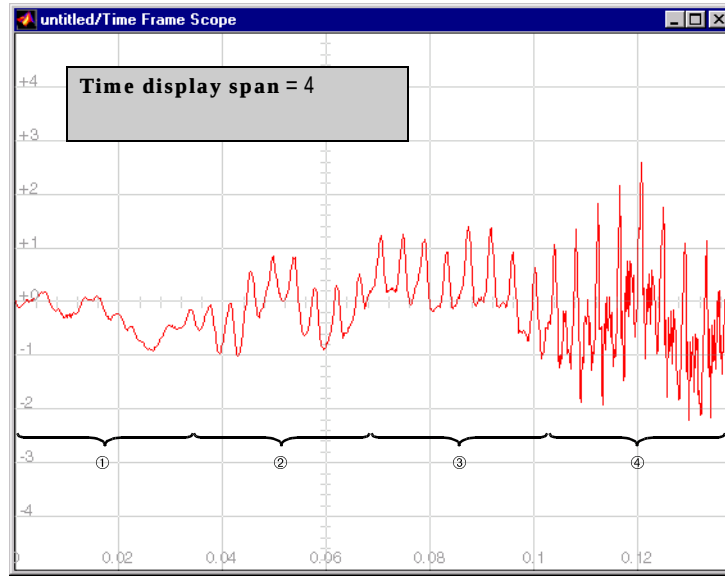
When displaying an M-by-N matrix containing time-domain data, the block assumes that each of the N input frames (columns) represent a succession of M consecutive samples taken from a time-series. That is, each data point in the input frame is assumed to correspond to a unique time value.

When displaying an N-by-M matrix of frequency-domain data, the block assumes that each of the N input frames (columns) is 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)`.

When displaying 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. The dialog box parameters give you complete freedom to plot the data in the most appropriate manner.

The scope updates the display for each new input frame. The number of sequential frames displayed on the scope is specified by the **Time display span** parameter for time-domain signals, and the **Horizontal display span** parameter for user-defined signals. Setting either parameter to 1 plots the current input frame's data across the entire width of the scope. Setting these display-span parameters to larger numbers allows you to see a broader section of the signal by fitting more frames of data into the display region. A single frame is the smallest unit that can be displayed, so neither parameter can be less than 1.



Scaling the Horizontal Axis for Time-Domain Signals

Scaling of the horizontal (time) axis for time-domain signals is automatic. The range of the time axis is $[0, S*T_{fi}]$, where T_{fi} is the input frame period, and S is the **Time display span** parameter. The spacing between time points is $T_{fi}/(M-1)$.

Scaling the Horizontal Axis for User-Defined Signals

To correctly scale the horizontal axis for user-defined signals, the block needs to know the spacing of the data in the input. This is specified by the **Increment per sample in input frame** parameter, I_S. This parameter represents the numerical interval between adjacent *x*-axis points corresponding to the input data. For example, an input signal sampled at 500 Hz has an increment per sample of 0.002 second. The actual units of this interval (seconds, meters, Volts, etc.) are not needed for axis scaling.

When the **Inherit sample increment from input** check box is selected, the block scales the horizontal axis by computing the horizontal interval between samples in the input frame from the frame period of the input. For example, if the input frame period is 1, and there are 64 samples per input frame, the interval between samples is computed to be 1/64. Computing the interval this way is usually only valid if the following conditions hold:

- The input is a nonoverlapping time-series; the *x*-axis on the scope represents time.
- The input's sample period (1/64 in the above example) is equal to the period with which the physical signal was originally sampled.

In other cases, the frame rate and frame size do not provide enough information for the block to correctly scale the horizontal axis, and you should specify the appropriate value for the **Increment per sample in input frame** parameter. The range of the horizontal axis is $[0,M*I_S*S]$, where M is the number of samples in each consecutive input frame, and S is the **Horizontal display span parameter**.

Scaling the Horizontal Axis for Frequency-Domain Signals

In order to correctly scale the horizontal (frequency) axis for frequency-domain signals, the Vector Scope block needs to know the sample period of the original time-domain sequence represented by the frequency-domain data. This is specified by the **Sample time of original time series** parameter.

When the **Inherit sample time from input** check box is selected, the block scales the frequency axis by reconstructing the frequency data from the frame-period of the frequency-domain input. This is valid when the following conditions hold:

- Each frame of frequency-domain data shares the same length as the frame of time-domain data from which it was generated; for example, when the FFT is computed on the same number of points as are contained in the time-domain input.
- The sample period of the time-domain signal in the simulation is equal to the period with which the physical signal was originally sampled.
- Consecutive frames containing the time-domain signal do not overlap each other; that is, a particular signal sample does not appear in more than one sequential frame.

In cases where not all of these conditions hold, you should specify the appropriate value for the **Sample time of original time-series** parameter.

The **Frequency units** parameter specifies whether the frequency axis values should be in units of **Hertz** or **rad/sec**, and the **Frequency range** parameter specifies the range of frequencies over which the magnitudes in the input should be plotted. The available options are **[0..Fs/2]**, **[-Fs/2..Fs/2]**, and **[0..Fs]**, where F_s is the original time-domain signal's sample frequency.

The Vector Scope block assumes that the input data spans the range $[0,F_s)$, as does the output from an FFT. To plot over the range [0..Fs/2] the scope truncates the input vector leaving only the first half of the data, then plots these remaining samples over half the frequency range. To plot over the range [-Fs/2..Fs/2], the scope reorders the input vector elements such that the last half of the data becomes the first half, and vice versa; then it relabels the *x*-axis accordingly.

If the **Frequency units** parameter specifies **Hertz**, the spacing between frequency points is $1/(M*T_s)$. For **Frequency units** of **rad/sec**, the spacing between frequency points is $2\pi/(M*T_s)$. The **Amplitude scaling** parameter allows you to select **Magnitude** or **dB** scaling along the *y*-axis.

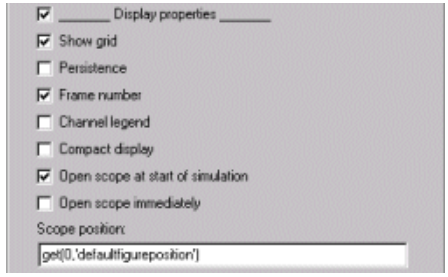
Scope Properties

The Vector Scope block allows you to plot time-domain, frequency-domain, or user-defined data, and adjust the frame span of the plot. Selecting the **Scope Properties** check box displays the **Input domain parameter**, which specifies the domain of the input data. In addition, for time-domain data, a **Time display span parameter** allows you to specify the number of frames to be displayed across the width of the scope window at any given time. For

user-defined data, a **Horizontal display span parameter** serves the same function. Both of these parameters must be 1 or greater. See “Displaying Data” on page 5-460 for more information.

Display Properties

The Vector Scope and Spectrum Scope blocks offer a similar collection of display property settings. These can be exposed in the parameter dialog box by selecting the **Display properties** check box. Many of the properties can be accessed under the **Axes** menu in the *unzoomed* scope view (when **Compact display** is deselected), or by right-clicking on the scope window.



The **Show grid** parameter toggles the background grid on and off. This option can also be set in the **Axes** menu of the scope window.

When **Persistence** is selected, the window maintains successive displays. That is, the scope does not erase the display after each frame (or collection of frames), but overlays successive input frames in the scope display. This option can also be set in the **Axes** menu of the scope window.

When **Frame number** is selected, the number of the current frame in the input sequence is displayed on the scope window, incrementing the count as each new input is received. Counting starts at 1 with the first input frame, and continues until the simulation stops.

When **Channel legend** is selected, a legend indicating the line color, style, and marker of each channel’s data is added. If the input signal is labeled, that label is displayed in the channel legend. If the input signal is not labeled, but comes from a Matrix Concatenation block with labeled inputs, those labels are displayed in the channel legend. Otherwise, each channel in the legend is labeled with the channel number (`CH 1`, `CH 2`, etc.). Click and drag on the legend

to reposition it in the scope window; double click on the line label to edit the text. Note that when the simulation is rerun, the new edits are lost and the labels revert to the defaults. The **Channel legend** option can also be set in the **Axes** menu of the scope window.

When **Compact display** is selected, the scope completely fills the containing figure window. Menus and axis titles are not displayed, and the numerical axis labels are shown within the axes. When **Compact display** is deselected, 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 deselected, the scope does not open automatically during the simulation. To view the scope, double-click on the Vector Scope block, which brings up the scope as well as the block parameter dialog box. This feature is useful when you have several scope blocks in a model, and you do not want to view all the associated scopes during the simulation.

Open scope immediately allows you to open the scope from the Vector Scope parameters dialog box while the simulation is running. If the simulation is running and the scope window is not visible, you can double-click on the scope block to expose the scope window and the parameters dialog box. If you close the scope window during simulation, you can make it visible again by checking the **Open scope immediately** check box as long as the simulation is running. The check box will become deselected as soon as the scope opens.

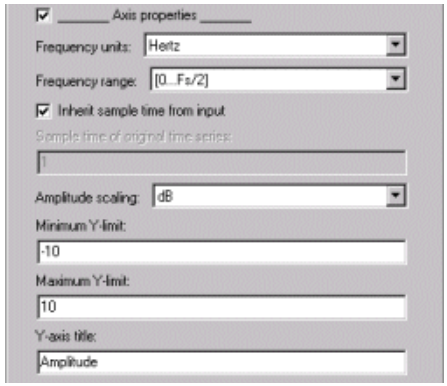
The **Scope position** parameter specifies a four-element vector of the form

```
[left bottom width height]
```

specifying the position of the scope window on the screen, where (0,0) is the lower-left corner of the display. See the MATLAB `figure` command for more information.

Axis Properties

The Vector Scope and Spectrum Scope blocks also share a similar collection of axis property settings. For the Vector Scope, the parameters listed under the **Axis properties** check box vary with the domain of the input. The dialogue box below shows the parameters available for frequency-domain data.



Minimum Y-limit and **Maximum Y-limit** set the range of the vertical axis. If **Autoscale** is selected from the right-click pop-up menu or from the **Axes** menu option, the **Minimum Y-limit** and **Maximum Y-limit** values are automatically recalculated to best fit the range of the data on the scope. Both of these parameters are available for all input domains.

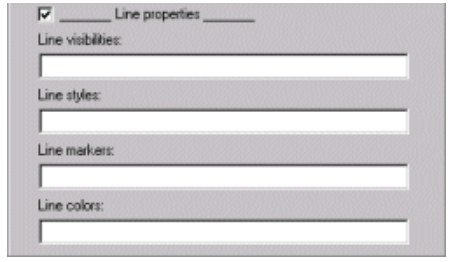
Y-axis title is the text to be displayed to the left of the *y*-axis. This parameter is available for all input domains. **X-axis title** is an analogous parameter available only when plotting user-defined data (this parameter is not visible in the dialog box shown).

Frequency-domain and user-defined data need extra information to scale the horizontal axis. For user-defined data, the parameters that provide this information are **Inherit sample increment from input** and **Increment in sample in input frame**. See “Scaling the Horizontal Axis for User-Defined Signals” on page 5-462 for more information. For frequency-domain data, an analogous pair of parameters, **Inherit sample time from input** and **Sample time of original time series**, must be specified. See “Scaling the Horizontal Axis for Frequency-Domain Signals” on page 5-462 for more information.

Three other parameters related to scaling the *x*-axis for frequency-domain signals are **Frequency units**, **Frequency range**, and **Amplitude scaling**. These are also described in “Scaling the Horizontal Axis for Frequency-Domain Signals” on page 5-462.

Line Properties

Both the Vector Scope and Spectrum scope also offer a similar collection of line property settings. These can be exposed in the parameter dialog box by selecting the **Line properties** check box. These properties can also be accessed under the **Channels** menu in the *unzoomed* scope view (when **Compact display** is deselected), or by right-clicking on the scope window.



The **Line properties** setting are typically used to help distinguish between two or more independent channels of data on the scope.

The **Line visibilities** parameter specifies which channels' data is displayed on the scope, and which is hidden. The syntax specifies the visibilities in list form, where the term `on` or `off` as a list entry specifies the visibility of the corresponding channel's data. The list entries are separated by the pipe symbol, `|`.

For example, a five-channel signal would ordinarily generate five distinct plots on the scope. To disable plotting of the third and fifth lines, enter the following visibility specification.

```
on | on | off | on | off
①    ②     ③     ④     ⑤
```

Note that the first (leftmost) list item corresponds to the first signal channel (leftmost column of the input matrix).

The **Line styles** parameter specifies the line style with which each channel's data is displayed on the scope. The syntax specifies the channel line styles in list form, with each list entry specifying a style for the corresponding channel's data. The list entries are separated by the pipe symbol, `|`.

For example, a five-channel signal would ordinarily generate all five plots with a solid line style. To instead plot each line with a different style, enter

```
- | -- | : | -. | -
①   ②    ③   ④    ⑤
```

These settings plot the signal channels with the following styles.

Line Style	Appearance
Solid	
Dashed	
Dotted	
Dash-dot	
Solid	

Note that the first (leftmost) list item, `'-'`, corresponds to the first signal channel (leftmost column of the input matrix). See `LineStyle` property of the `line` function in the MATLAB documentation for more information about the style syntax. To specify a marker for the individual sample points, use the **Line markers** parameter, described below.

The **Line markers** parameter specifies the marker style with which each channel's samples are represented on the scope. The syntax specifies the channels' marker styles in list form, with each list entry specifying a marker for the corresponding channel's data. The list entries are separated by the pipe symbol, `|`.

For example, a five-channel signal would ordinarily generate all five plots with no marker symbol (i.e., the individual sample points are not marked on the scope). To instead plot each line with a different marker style, you could enter

```
* | . | x | s | d
①   ②   ③   ④   ⑤
```

These settings plot the signal channels with the following styles.

Marker Style	Appearance
Asterisk	
Point	
Cross	
Square	
Diamond	

Note that the first (leftmost) list item, `'*'`, corresponds to the first signal channel (leftmost column of the input matrix). See the `Marker` property of the `line` function in the MATLAB documentaion for more information about the available markers.

Type the word `stem` instead of one of the basic `Marker` shapes to produce a *stem plot* for the data in a particular channel.

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 MATLAB's `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]
   ①         ②         ③          ④             ⑤
```

or

```
'k' | 'b' | 'r' | 'g' | [.7529 0 .7529]
 ①     ②     ③     ④           ⑤
```

These settings plot the signal channels in the following colors (8-bit RGB equivalents shown in the center column).

Color	RGB Equivalent	Appearance
Black	(0,0,0)	
Blue	(0,0,255)	
Red	(255,0,0)	
Green	(0,255,0)	
Dark purple	(192,0,192)	

Note that the first (leftmost) list item, `'k'`, corresponds to the first signal channel (leftmost column of the input matrix). See `ColorSpec` in the online MATLAB documentaion for more information about the color syntax.

Scope Window

The scope title (in the window title bar) is the same as the block title. The axis scaling is set by parameters listed under the **Axis properties** check box in the dialog box.

In addition to the standard MATLAB figure window menus (**File**, **Edit**, **Window**, **Help**), the Vector Scope window has an **Axes** and a **Channels** menu.

The properties listed in the **Axes** menu apply to all channels. Many of the parameters in this menu are also accessible through the block parameter dialog box. These are **Persistence**, **Show grid**, **Compact display**, **Frame number**, and **Channel legend**; see “Display Properties” on page 5-464 for more information. Below are descriptions of the other parameters listed in the **Axes** menu:

- **Refresh** erases all data on the scope display, except for the most recent trace. This command is useful in conjunction with the **Persistence** setting.
- **Autoscale** resizes the *y*-axis to best fit the vertical range of the data. The numerical limits selected by the autoscale feature are displayed in the **Minimum Y-limit** and **Maximum Y-limit** parameters in the parameter dialog box. You can change them by editing those values.

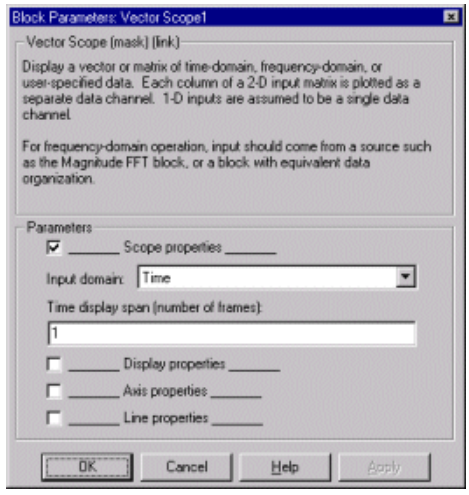
- **Save Position** automatically updates the **Scope position** parameter in the **Axis properties** field to reflect the scope window's current position and size. To make the scope window open at a particular location on the screen when the simulation runs, simply drag the window to the desired location, resize it as needed, and select **Save Position**. Note that the parameter dialog box must be closed when you select **Save Position** in order for the **Scope position** parameter to be updated.

The properties listed in the **Channels** menu apply to a particular channel. The parameters listed in this menu are **Visible**, **Style**, **Marker**, and **Color**; they correspond to the parameters listed in the dialog box under the **Line properties** check box. See "Line Properties" on page 5-467 for more information.

Many of these options can also be accessed by right-clicking with the mouse anywhere on the scope display. The menu that pops up contains a combination of the options available in both the **Axes** and **Channels** menus. The right-click menu is very helpful when the scope is in zoomed mode, when the **Axes** and **Channels** menus are not visible.

Dialog Box

Scope Properties Dialog Box



Scope properties

Select to expose **Scope properties** panel.

Input domain

The domain of the input; **Time**, **Frequency**, or **User-defined**.

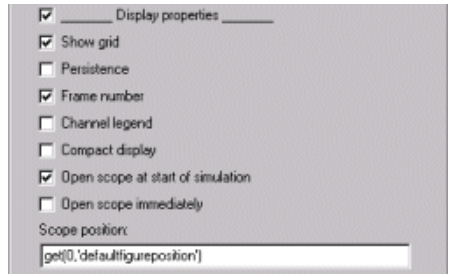
Time display span

The number of consecutive frames to display (horizontally) on the scope at any one time. (Visible when the **Input domain** parameter is **Time**.)

Horizontal display span

(Not visible in the dialog box shown; appears under **Scope properties** when the **Input domain** parameter is **User-defined**.) The number of consecutive frames to display (horizontally) on the scope at any one time.

Display Properties Dialog Box



Display properties

Select to expose **Display properties** panel.

Show grid

Toggles the scope grid on and off. 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. Tunable.

Frame number

Displays the number of the current frame in the input sequence, when selected with **Compact display** off. The frame number is not shown when **Compact display** is selected. Tunable.

Channel legend

Toggles the legend on and off. Tunable.

Compact display

Resizes the scope to fill the window. Tunable.

Open scope at start of simulation

Opens the scope at the start of the simulation. When this parameter is deselected, the scope will not open automatically during the simulation; to view the scope, double click on the Vector Scope block during the simulation. This will bring up the scope as well as the block parameter dialog box.

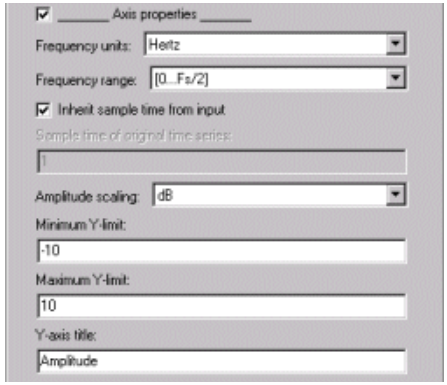
Open scope immediately

Opens the scope from the Vector Scope parameters dialog box while the simulation is running. The check box becomes deselected automatically after use.

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.

Axis properties Dialog Box



Axis properties

Select to expose the **Axis Properties** panel. Tunable.

Frequency units

The frequency units for the *x*-axis, **Hertz** or **rad/sec**. (Visible when the **Input domain** parameter is **Frequency**.) Tunable.

Frequency range

The frequency range over which to plot the data, **[0..Fs/2]**, **[-Fs/2..Fs/2]**, or **[0..Fs]**, where F_s is the sample frequency of the original time-domain signal, $1/T_s$. (Visible when the **Input domain** parameter is **Frequency**.) Tunable.

Inherit sample time from input

Computes the time-domain sample period from the frame period and frame size of the frequency-domain input; use only if the length of the each frame

of frequency-domain data is the same as the length of the frame of time-domain data from which is was generated. (Visible when the **Input domain** parameter is **Frequency**.) Tunable.

Sample time of original time series

The sample period of the original time-domain signal, T_S. (Visible when the **Input domain** parameter is **Frequency**.) Tunable.

Inherit sample increment from input

(Not visible in the dialog box shown; appears under **Axis properties** when the **Input domain** parameter is **User-defined**.) Scales the horizontal axis by computing the horizontal interval between samples in the input frame from the frame period of the input; use only if the input's sample period is equal to the period with which the physical signal was originally sampled. Tunable.

Increment per sample in input frame

(Not visible in the dialog box shown; appears under **Axis properties** when the **Input domain** parameter is **User-defined**.) The numerical interval between adjacent *x*-axis points corresponding to the user-defined input data. Tunable.

Amplitude scaling

The scaling for the *y*-axis, **dB** or **Magnitude**. (Visible when the **Input domain** parameter is **Frequency**.) Tunable.

Minimum Y-limit

The minimum value of the *y*-axis.

Maximum Y-limit

The maximum value of the *y*-axis

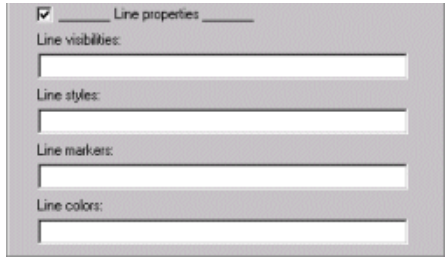
Y-Axis title

The text to be displayed to the left of the *y*-axis.

X-Axis title

(Not visible in the dialog box shown; appears under **Axis properties** when the **Input domain** parameter is **User-defined**.) The text to be displayed below the *x*-axis.

Line Properties Dialog Box



Line properties

Select to expose the **Line Properties** panel. Tunable.

Line visibilities

The visibility of the various channels' scope traces, `on` or `off`. Channels are separated by a pipe (|) symbol. Tunable.

Line styles

The line styles of the various channels' scope traces. Channels are separated by a pipe (|) symbol. Tunable.

Line markers

The line markers of the various channels' scope traces. Channels are separated by a pipe (|) symbol. Tunable.

Line colors

The colors of the various channels' scope traces, in one of the `ColorSpec` formats. Channels are separated by a pipe (|) symbol. Tunable.

See Also

Matrix Viewer	DSP Blockset
Spectrum Scope	DSP Blockset

See "Viewing Signals" on page 3-80 for related information.

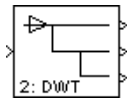
Purpose

Decompose a signal into components of logarithmically decreasing frequency intervals and sample rates (requires the Wavelet Toolbox).

Library

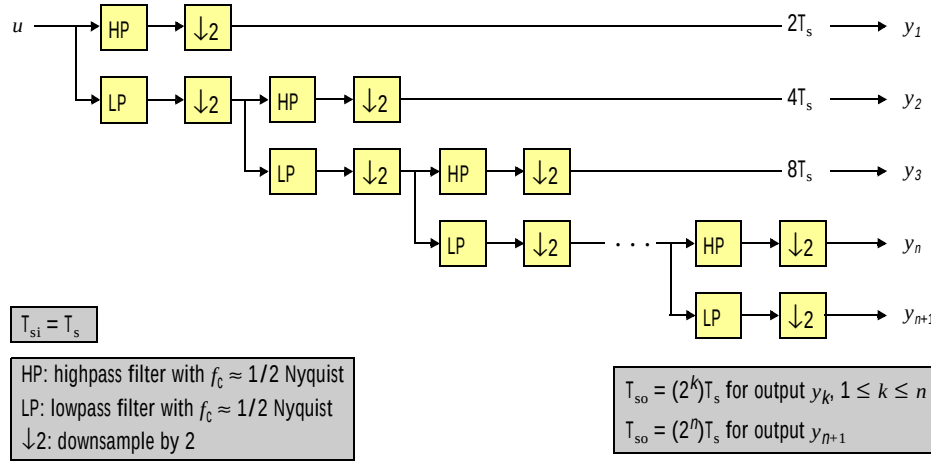
Filtering / Multirate Filters

Description



The Wavelet Analysis block uses the `wfilters` function from the Wavelet Toolbox to construct a dyadic analysis filter bank that decomposes a broadband signal into a collection of successively more bandlimited components. An *n*-level filter bank structure is shown below, where *n* is specified by the **Number of levels** parameter.

Wavelet Analysis Filter Bank, n Levels



At each level, the *low-frequency* output of the previous level is decomposed into adjacent high- and low-frequency subbands by a highpass (HP) and lowpass (LP) filter pair. Each of the two output subbands is half the bandwidth of the input to that level. The bandlimited output of each filter is maximally decimated by a factor of 2 to preserve the bit rate of the original signal.

Filter Coefficients

The filter coefficients for the highpass and lowpass filters are computed by the Wavelet Toolbox function `wfilters`, based on the wavelet specified in the **Wavelet name** parameter. The table below lists the available options.

Wavelet Name	Sample Wavelet Function Syntax
Haar	`wfilters('haar')`
Daubechies	`wfilters('db4')`
Symlets	`wfilters('sym3')`
Coiflets	`wfilters('coif1')`
Biorthogonal	`wfilters('bior3.1')`
Reverse Biorthogonal	`wfilters('rbio3.1')`
Discrete Meyer	`wfilters('dmey')`

The **Daubechies**, **Symlets**, and **Coiflets** options enable a secondary **Wavelet order** parameter that allows you to specify the wavelet order. For example, if you specify a **Daubechies** wavelet with **Wavelet order** equal to 6, the Wavelet Analysis block calls the `wfilters` function with input argument `'db6'`.

The **Biorthogonal** and **Reverse Biorthogonal** options enable a secondary **Filter order [synthesis / analysis]** parameter that allows you to independently specify the wavelet order for the analysis and synthesis filter stages. For example, if you specify a **Biorthogonal** wavelet with **Filter order [synthesis / analysis]** equal to `[2 / 6]`, the Wavelet Analysis block calls the `wfilters` function with input argument `'bior2.6'`.

See the Wavelet Toolbox decantation for more information about the `wfilters` function. If you want to explicitly specify the FIR coefficients for the analysis filter bank, use the Dyadic Analysis Filter Bank block.

Tree Structure

The wavelet tree structure has n+1 outputs, where n is the number of levels. The sample rate and bandwidth of the top output are half the input sample rate and bandwidth. The sample rate and bandwidth of each additional output

(except the last) are half that of the output from the previous level. In general, for an input with sample period $T_{si} = T_s$, and bandwidth BW, output y_k has sample period $T_{so,k}$ and bandwidth BW_k.

$$T_{so,k} = \begin{cases} (2^k)T_s & (1 \le k \le n) \\ (2^n)T_s & (k = n+1) \end{cases}$$

$$BW_k = \begin{cases} \frac{BW}{2^k} & (1 \le k \le n) \\ \frac{BW}{2^n} & (k = n+1) \end{cases}$$

Note that in frame-based mode, the change in the sample period of output y_k is reflected by its frame size, $M_{o,k}$, rather than by its frame rate.

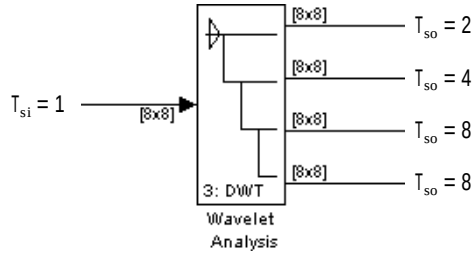
$$M_{o,k} = \begin{cases} \frac{M_i}{2^k} & (1 \le k \le n) \\ \frac{M_i}{2^n} & (k = n+1) \end{cases}$$

The bottom two outputs (y_n and y_{n+1}) share the same sample period, bandwidth, and frame size because they originate at the same tree level.

Sample-Based Operation

An M-by-N sample-based matrix input is treated as M∗N independent channels, and the block filters each channel independently over time. The output at each port is the same size as the input, one output channel for each input channel. As described earlier, each output port has a different sample period.

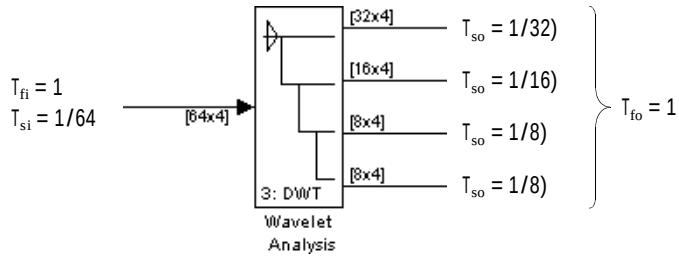
The figure below shows the input and output sample periods for a 64-channel sample-based input to a three-level filter bank. The input has a period of 1, so the fastest output has a period of 2.



Frame-Based Operation

An M_i-by-N frame-based matrix input is treated as N independent channels, and the block filters each channel independently over time. The input frame size M_i must be a multiple of 2^n, and *n* is the number of filter bank levels. For example, a frame size of 8 would be appropriate for a three-level tree (2^3=8). The number of columns in each output is the same as the number of columns in the input.

Each output port has the *same frame period* as the input. The reduction in the output sample rates results from the smaller output frame sizes, as shown in the example below for a four-channel input to a three-level filter bank.



Latency

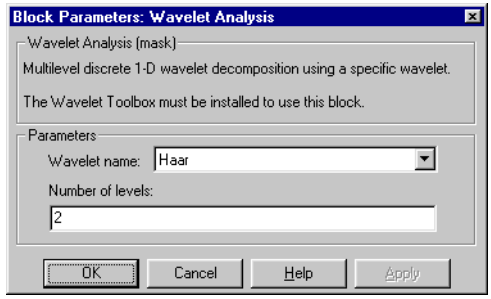
Zero Latency. The Wavelet Analysis block has *no tasking latency* for frame-based operation, which is always single-rate. The block therefore analyzes the first input sample (received at *t*=0) to produce the first output sample at each port.

Nonzero Latency. For sample-based operation, the Wavelet Analysis block is multirate and has 2^{n-1} samples of latency in both Simulink tasking modes. As a result, the block repeats a zero initial condition in each channel for the first

2^{n-1} output samples, before propagating the first analyzed input sample (computed from the input received at t=0).

See “Excess Algorithmic Delay (Tasking Latency)” on page 3-91 and “The Simulation Parameters Dialog Box” in the Simulink documentation for more information about block rates and Simulink's tasking modes.

Dialog Box



The parameters displayed in the dialog box vary for different wavelet types. Only some of the parameters listed below are visible in the dialog box at any one time.

Wavelet name

The wavelet used in the analysis.

Wavelet order

The order for the **Daubechies**, **Symlets**, and **Coiflets** wavelets. This parameter is available only when one of these wavelets is selected in the **Wavelet name** menu.

Filter order [synthesis / analysis]

The filter orders for the synthesis and analysis stages of the **Biorthogonal** and **Reverse Biorthogonal** wavelets. For example, `[2 / 6]` selects a second-order synthesis stage and a sixth-order analysis stage. The **Filter order** parameter is available only when one of the above wavelets is selected in the **Wavelet name** menu.

Number of levels

The number of filter bank levels. An n-level structure has n+1 outputs.

References

Fliege, N. J. *Multirate Digital Signal Processing: Multirate Systems, Filter Banks, Wavelets*. West Sussex, England: John Wiley & Sons, 1994.

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.

See Also

Dyadic Analysis Filter Bank	DSP Blockset
Wavelet Synthesis	DSP Blockset
`wfilters`	Wavelet Toolbox

See the following sections for related information:

- "Converting Sample Rates and Frame Rates" on page 3-20
- "Multirate Filters" on page 4-24

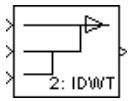
Purpose

Reconstruct a signal from its multirate bandlimited components (requires the Wavelet Toolbox).

Library

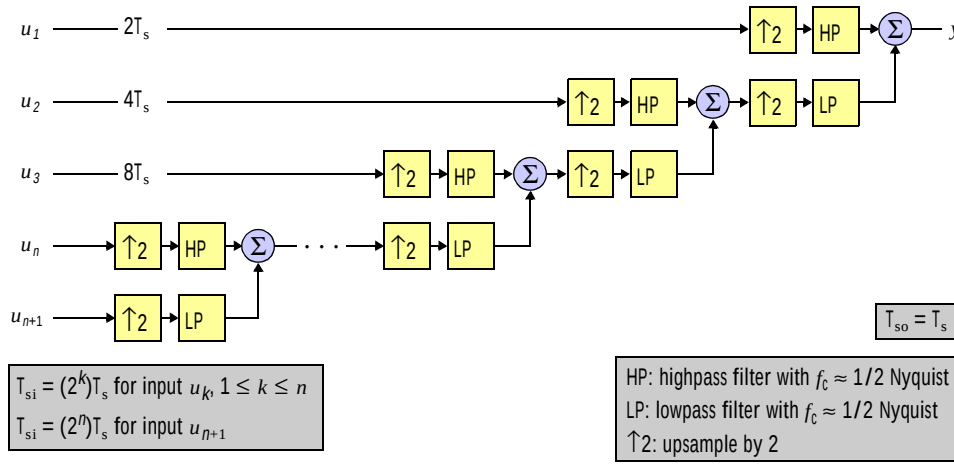
Filtering / Multirate Filters

Description



The Wavelet Synthesis block uses the `wfilters` function from the Wavelet Toolbox to reconstruct a signal that was decomposed by the Wavelet Analysis block. The reconstruction or *synthesis* process is the inverse of the analysis process, and restores the original signal by upsampling, filtering, and summing the bandlimited inputs in stages corresponding to the analysis process. An *n*-level synthesis filter bank structure is shown below, where *n* is specified by the **Number of levels** parameter.

Wavelet Synthesis Filter Bank, n Levels



At each level, the two bandlimited inputs (one low-frequency, one high-frequency, both with the same sample rate) are upsampled by a factor of 2 to match the sample rate of the input to the next stage. They are then filtered by a highpass (HP) and lowpass (LP) filter pair with coefficients calculated to cancel (in the subsequent summation) the aliasing introduced in the corresponding analysis filter stage. The output from each (upsample-filter-sum) level has twice the bandwidth and twice the sample rate of the input to that level.

For perfect reconstruction, the Wavelet Synthesis and Wavelet Analysis blocks must have the same parameter settings.

Filter Coefficients

The filter coefficients for the highpass and lowpass filters are computed by the Wavelet Toolbox function `wfilters`, based on the wavelet specified in the **Wavelet name** parameter. The table below lists the available options.

Wavelet Name	Sample Wavelet Function Syntax
Haar	`wfilters('haar')`
Daubechies	`wfilters('db4')`
Symlets	`wfilters('sym3')`
Coiflets	`wfilters('coif1')`
Biorthogonal	`wfilters('bior3.1')`
Reverse Biorthogonal	`wfilters('rbio3.1')`
Discrete Meyer	`wfilters('dmey')`

The **Daubechies**, **Symlets**, and **Coiflets** options enable a secondary **Wavelet order** parameter that allows you to specify the wavelet order. For example, if you specify a **Daubechies** wavelet with **Wavelet order** equal to 6, the Wavelet Synthesis block calls the `wfilters` function with input argument `'db6'`.

The **Biorthogonal** and **Reverse Biorthogonal** options enable a secondary **Filter order [synthesis / analysis]** parameter that allows you to independently specify the wavelet order for the analysis and synthesis filter stages. For example, if you specify a **Biorthogonal** wavelet with **Filter order [synthesis / analysis]** equal to `[2 / 6]`, the Wavelet Synthesis block calls the `wfilters` function with input argument `'bior2.6'`.

See the Wavelet Toolbox documentation for more information about the `wfilters` function. If you want to explicitly specify the FIR coefficients for the synthesis filter bank, use the Dyadic Synthesis Filter Bank block.

Tree Structure

The wavelet tree structure has n+1 inputs, where n is the number of levels. The sample rate and bandwidth of the output are twice the sample rate and bandwidth of the top input. The sample rate and bandwidth of each additional input (except the last) are half that of the input to the previous level.

$$T_{si,\,k+1} = 2T_{si,\,k} \qquad 1 \le k < n$$

$$BW_{k+1} = \frac{BW_k}{2} \qquad 1 \le k < n$$

The bottom two inputs (u_n and u_{n+1}) should have the same sample rate and bandwidth since they are processed by the same level.

$$T_{si,\,n+1} = T_{si,\,n}$$

$$BW_{n+1} = BW_n$$

Note that in frame-based mode, the sample period of input u_k is reflected by its frame size, $M_{i,k}$, rather than by its frame rate.

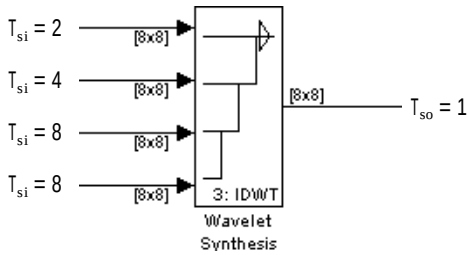
$$M_{i,\,k+1} = \frac{M_{i,\,k}}{2} \qquad 1 \le k < n$$

$$M_{i,\,n+1} = M_{i,\,n}$$

Sample-Based Operation

An M-by-N sample-based matrix input is treated as M∗N independent channels, and the block filters each channel independently over time. The output is the same size as the input at each port, one output channel for each input channel. As described earlier, each input port has a different sample period.

The figure below shows the input and output sample periods for the four 64-channel sample-based inputs to a three-level filter bank. The fastest input has a period of 2, so the output period is 1.

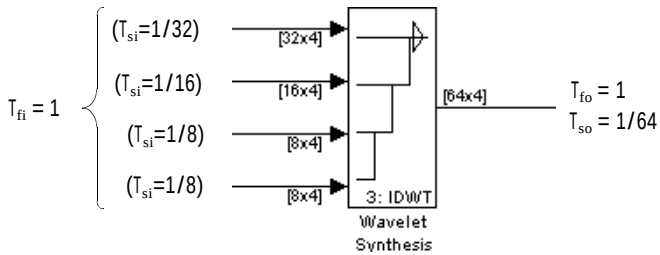


Frame-Based Operation

An M_i-by-N frame-based matrix input is treated as N independent channels, and the block filters each channel independently over time. The number of columns in the output is the same as the number of columns in the input.

All inputs must have the same frame period, which is also the output frame period. The different input sample rates should be represented by the input frame sizes: If the input to the top port has frame size M_i, the input to the second-from-top port should have frame size $M_i/2$, the input to the third-from-top port should have frame size $M_i/4$, and so on. The input to the *bottom* port should have the same frame size as the second-from-bottom port. The increase in the sample rate of the output is also represented by its frame size, which is twice the largest input frame size.

The relationship between sample periods, frame periods, and frame sizes is shown below for a four-channel frame-based input to a 3-level filter bank.



Latency

Zero Latency. The Wavelet Synthesis block has *no tasking latency* for frame-based operation, which is always single-rate. The block therefore uses the first input samples (received at t=0) to synthesize the first output sample.

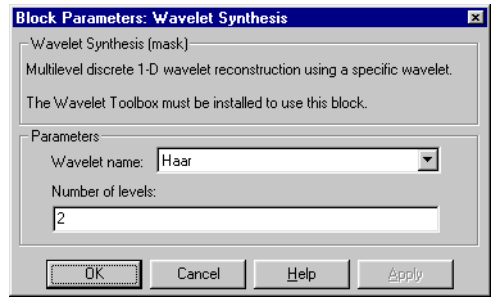
Nonzero Latency. For sample-based operation, the Wavelet Synthesis block is multirate and has the following tasking latencies:

- 2^n-2 samples in Simulink's single-tasking mode
- 2^n samples in Simulink's multitasking mode

In the above cases, the block repeats a zero initial condition in each channel for the first D output samples, where D is the latency shown above. For example, in single-tasking mode the block generates 2^n-2 zero-valued output samples in each channel before propagating the first synthesized output sample (computed from the inputs received at t=0).

See "Excess Algorithmic Delay (Tasking Latency)" on page 3-91 and "The Simulation Parameters Dialog Box" in the Simulink documentation for more information about block rates and Simulink's tasking modes.

Dialog Box



The parameters displayed in the dialog box vary for different wavelet types. Only some of the parameters listed below are visible in the dialog box at any one time.

Wavelet name

The wavelet used in the synthesis.

Wavelet order

The order for the **Daubechies**, **Symlets**, and **Coiflets** wavelets. This parameter is available only when one of these wavelets is selected in the **Wavelet name** menu.

Filter order [synthesis / analysis]

The filter orders for the synthesis and analysis stages of the **Biorthogonal** and **Reverse Biorthogonal** wavelets. For example, `[2 / 6]` selects a second-order synthesis stage and a sixth-order analysis stage. The **Filter order** parameter is available only when one of the above wavelets is selected in the **Wavelet name** menu.

Number of levels

The number of filter bank levels. An *n*-level structure has *n*+1 outputs.

References

Fliege, N. J. *Multirate Digital Signal Processing: Multirate Systems, Filter Banks, Wavelets*. West Sussex, England: John Wiley & Sons, 1994.

Strang, G. and T. Nguyen. *Wavelets and Filter Banks*. Wellesley, MA: Wellesley-Cambridge Press, 1996.

Vaidyanathan, P. P. *Multirate Systems and Filter Banks*. Englewood Cliffs, NJ: Prentice Hall, 1993.

See Also

Dyadic Synthesis Filter Bank	DSP Blockset
Wavelet Analysis	DSP Blockset
`wfilters`	Wavelet Toolbox

See the following sections for related information:

- "Converting Sample Rates and Frame Rates" on page 3-20
- "Multirate Filters" on page 4-24

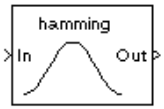
Purpose

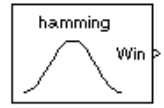
Compute a window, and/or apply a window to an input signal.

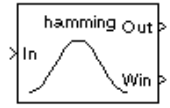
Library

DSP Sources, Signal Operations

Description







The Window Function block has three modes of operation, selected by the **Operation** parameter as described below.

Operation Modes

In each mode, the block first creates a window vector, *w*, by sampling the window specified in the **Window type** parameter at M discrete points. The **Operation** modes are:

- **Apply window to input**

  In this mode the block computes an M-by-1 window vector, *w*, and multiplies the vector element-wise with each of the N channels in the M-by-N input matrix *u*.

  ```
  y = repmat(w,1,N) .* u          % Equivalent MATLAB code
  ```

  A length-M 1-D vector input is treated as an M-by-1 matrix. The output, *y*, always has the same dimension as the input. If the input is frame-based, the output is frame-based; otherwise, the output is sample-based.

- **Generate window**

  In this mode the block generates a sample-based 1-D window vector, *w*, with length M specified by the **Window length** parameter. The `In` port is disabled.

- **Generate and apply window**

  In this mode the block computes an M-by-1 window vector, *w*, and multiplies the vector element-wise with each of the N channels in the M-by-N input matrix *u*.

  ```
  y = repmat(w,1,N) .* u          % Equivalent MATLAB code
  ```

  A length-M 1-D vector input is treated as an M-by-1 matrix. The block produces two outputs:

  - At the `Out` port, the block produces the result of the multiplication, *y*, which has the same dimension as the input. If the input is frame-based, output *y* is frame-based; otherwise, output *y* is sample-based.
  - At the `Win` port, the block produces the M-by-1 window vector, *w*. Output *w* is always sample-based.

Window Sampling

For the generalized-cosine windows (**Blackman**, **Hamming**, and **Hann**), the **Sampling** parameter determines whether the window samples are computed in a *periodic* or a *symmetric* manner. For example, if **Sampling** is set to **Symmetric**, a Hamming window of length M is computed as

```
w = hamming(M)      % Symmetric (aperiodic) window
```

If **Sampling** is set to **Periodic**, the same window is computed as

```
w = hamming(M+1)    % Periodic (asymmetric) window
w = w(1:M)
```

Window Type

The available window types are shown in the table below. The **Stopband attenuation in dB** and **Beta** parameters specify the characteristics of the **Chebyshev** and **Kaiser** windows, respectively, and are only available when those window designs are selected.

When **Window type** is set to **User defined**, the Window function block computes the user-defined window specified by the **Window function name** parameter. If the user-defined window requires parameters other than the window length, select the **Additional parameters for user defined window** check box. The cell array entered in **Window function parameters** determines the values of the additional parameters.

For complete information about the other window functions, consult the Signal Processing Toolbox documentation.

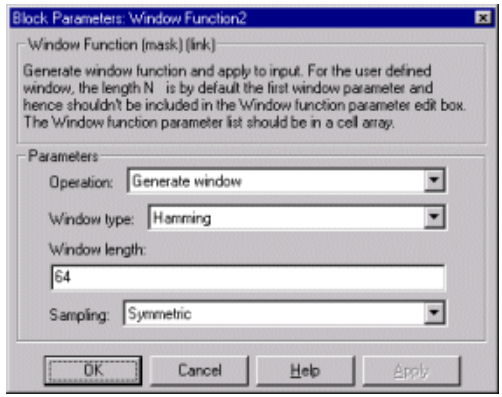
Window Type	Description
Bartlett	Computes a Bartlett window. `w = bartlett(M)`
Blackman	Computes a Blackman window. `w = blackman(M)`
Boxcar	Computes a Boxcar window. `w = boxcar(M)`

Window Type	Description
Chebyshev	Computes a Chebyshev window with stopband ripple R. `w = chebwin(M,R)`
Hamming	Computes a Hamming window. `w = hamming(M)`
Hann	Computes a Hann window (also known as a Hanning window). `w = hann(M)`
Hanning	Obsolete. This window option is included only for compatibility with older models. Use the **Hann** option instead of **Hanning** whenever possible.
Kaiser	Computes a Kaiser window with Kaiser parameter beta. `w = kaiser(M,beta)`
Triang	Computes a triangular window. `w = triang(M)`

User Defined	Computes the user-defined window function specified by the entry in the **Window function name** parameter, usrwin.
	`w = usrwin(M) % window takes no extra parameters`
	`w = usrwin(M,`x_1`,...,`x_n`) % window takes extra parameters {`x_1 ... x_n`}`

Window Function

Dialog Box



Operation

The block's operation: **Apply window to input**, **Generate window**, or **Generate and apply window**. The input/output port configuration is updated to match the parameter setting.

Window type

The type of window to apply. Tunable.

Window length

The length of the window to apply. This parameter is available only when **Generate window** is selected in the **Operation** menu. Otherwise, the window vector length is computed to match the input frame size, M.

Sampling

The window sampling for generalized-cosine windows, **Symmetric** or **Periodic**. Tunable.

Stopband attenuation in dB

(Not shown in dialog above. Visible for the **Chebyshev** window.) The level (dB) of stopband attenuation, R_s. Tunable.

Beta

(Not shown in dialog above. Visible for the **Kaiser** window.) The **Kaiser** window β parameter. Increasing β widens the mainlobe and decreases the amplitude of the window sidelobes in the window's frequency magnitude response. Tunable.

Window function name

(Not shown in dialog above. Visible for **User defined** windows.) The name of the user-defined window function to be calculated by the block. Tunable.

Additional parameters for user defined window

(Not shown in dialog above. Visible for **User defined** windows.) Enables the **Window function parameters** wnen selected. Select when the user-defined window requires parameters other than the window length. Tunable.

Window function parameters

(Not shown in dialog above.Visible for **User defined** windows.) The extra parameters required by the user-defined window function, enabled when **Additional parameters for user defined window** is selected. The entry must be a cell array. Tunable.

See Also

FFT	DSP Blockset
`bartlett`	Signal Processing Toolbox
`blackman`	Signal Processing Toolbox
`boxcar`	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

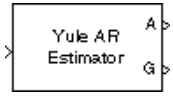
Purpose

Compute an estimate of AR model parameters using the Yule-Walker method.

Library

Estimation / Parametric Estimation

Description



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.

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.

$$H(z) = \frac{G}{A(z)} = \frac{G}{1 + a(2)z^{-1} + \ldots + a(p+1)z^{-p}}$$

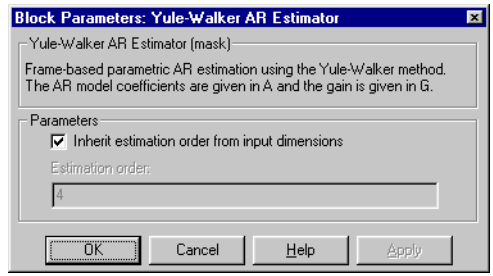
When **Inherit estimation order from input dimensions** is selected, the order, p, of the all-pole model is one less that the length of the input vector. Otherwise, the order is the value specified by the **Estimation order** parameter. The Yule-Walker AR Estimator and Burg AR Estimator blocks return similar results for large frame sizes.

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



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.

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 IIR Filter Design

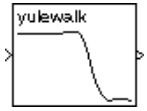
Purpose

Design and apply an IIR filter.

Library

Filtering / Filter Designs

Description



The Yule-Walker IIR Filter Design block designs a recursive (ARMA) digital filter with arbitrary multiband magnitude response, and applies it to a discrete-time input using the Direct-Form II Transpose Filter block. The filter design, which uses the `yulewalk` function in the Signal Processing Toolbox, performs a least-squares fit to the specified frequency response.

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

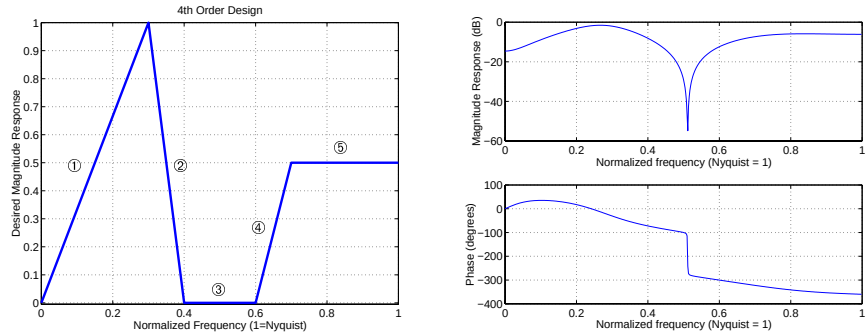
The **Band-edge frequency vector** parameter is a vector of frequency points in the range `0` to `1`, where `1` corresponds to half the sample frequency. The first element of this vector must be `0` and the last element `1`, and intermediate points must appear in ascending order. The **Magnitudes at these frequencies** parameter is a vector containing the desired magnitude response at the points specified in the **Band-edge frequency vector**.

Note that, unlike the Remez FIR Filter Design block, each frequency-magnitude pair specifies the junction of two adjacent frequency bands, so there are no “don’t care” regions.

Band edge frequency = `[0.0 0.3 0.4 0.6 0.7 1.0]`

Magnitudes = `[0.0 1.0 0.0 0.0 0.5 0.5]`

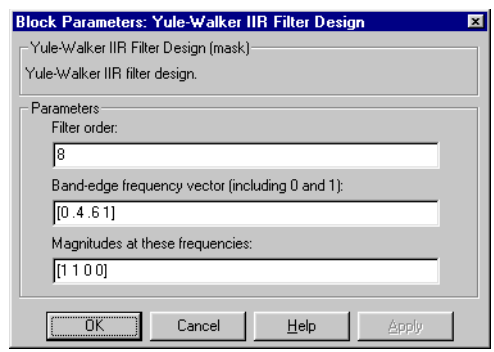
Band: ① ② ③ ④ ⑤



When specifying the **Band-edge frequency vector** and **Magnitudes at these frequencies** vectors, avoid excessively sharp transitions from passband to stopband. You may need to experiment with the slope of the transition region to get the best filter design.

For more details on the Yule-Walker filter design algorithm, see the description of the `yulewalk` function in the Signal Processing Toolbox documentation.

Dialog Box



Filter order

The order of the filter.

Band-edge frequency vector

A vector of frequency points. The value `1` corresponds to half the sample frequency. The first element of this vector must be `0` and the last element `1`. Tunable.

Magnitudes at these frequencies

A vector of frequency response magnitudes corresponding to the points in the **Band-edge frequency vector**. This vector must be the same length as the **Band-edge frequency vector**. Tunable.

References

Oppenheim, A. V. and R. W. Schafer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989.

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

See Also

Digital IIR Filter Design	DSP Blockset
Least Squares FIR Filter Design	DSP Blockset
Remez FIR Filter Design	DSP Blockset
`yulewalk`	Signal Processing Toolbox

See “Filter Designs” on page 4-3 for related information.

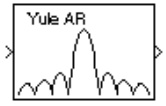
Purpose

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.

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.

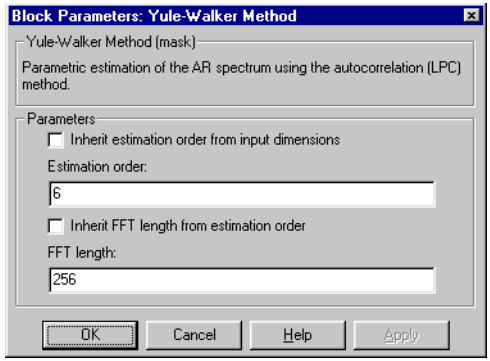
When **Inherit estimation order from input dimensions** is selected, the order of the all-pole model is one less that the input frame size. Otherwise, the order is the value specified by the **Estimation order** parameter. The spectrum is computed from the FFT of the estimated AR model parameters.

When **Inherit FFT length from 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. The Yule-Walker Method and Burg Method blocks return similar results for large buffer lengths.

Yule-Walker Method

Dialog Box



Inherit estimation order from input dimensions

When selected, sets the estimation order to one less than the length of the input vector.

Estimation order

The order of the AR model. This parameter is enabled when **Inherit estimation order from input dimensions** is not selected.

Inherit FFT length from 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.

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.

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 4-30 for related information.

Zero Pad

Purpose

Alter the input size by zero-padding or truncating rows and/or columns.

Library

Signal Operations

Description

The Zero Pad block changes the size of the input matrix from M_i-by-N_i to M_o-by-N_o by zero-padding or truncating along the rows, the columns, or both dimensions. The dimensions of the output, M_o and N_o, are specified by the **Number of output rows** and **Number of output columns** parameters, respectively.

The **Zero pad along** parameter specifies how the input should be altered. The options are:

- **Columns**

  When **Columns** is selected, the **Number of output rows** parameter (M_o) is enabled, and the block pads or truncates each input *column* by an equal amount. If $M_o>M_i$, the block pads by adding M_o-M_i rows of zeros to the bottom of the matrix. If $M_o<M_i$, the block truncates by deleting M_i-M_o rows from the bottom of the matrix. In both cases, the number of columns is unchanged ($N_o=N_i$). A 1-D vector input is zero padded or truncated at the "bottom," and the output is a 1-D vector.

- **Rows**

  When **Rows** is selected, the **Number of output columns** parameter (N_o) is enabled, and the block pads or truncates each input *row* by an equal amount. If $N_o>N_i$, the block pads by adding N_o-N_i columns of zeros to the right side of the matrix. If $N_o<N_i$, the block truncates by deleting N_i-N_o columns from the right side of the matrix. In both cases, the number of rows is unchanged ($M_o=M_i$). A 1-D vector input is zero padded or truncated at the "bottom," and the output is a 1-D vector.

- **Columns and rows**

  When **Columns and rows** is selected, both the **Number of output rows** parameter (M_o) and the **Number of output columns** parameter (N_o) are enabled, and the block pads or truncates rows and columns as specified. A length-M_i 1-D vector input is treated as an M_i-by-1 matrix and the output is an M_o-by-N_o matrix.

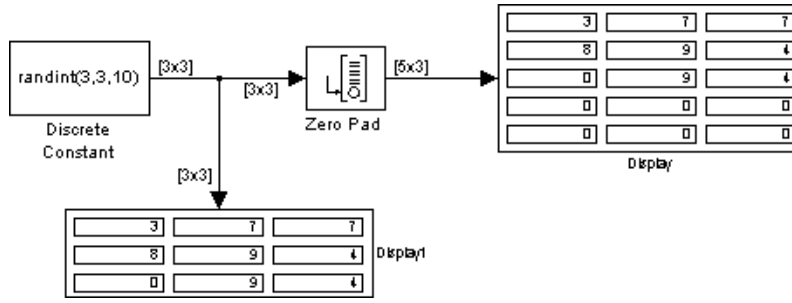
- **None**

  When **None** is selected, the input is passed through to the output without padding or truncation.

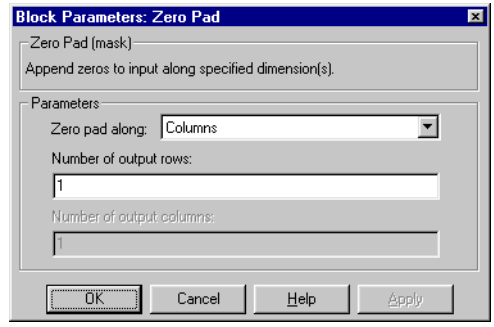
Example

In the model below, the 3-by-3 input is zero-padded along the column dimension to 5-by-3. The parameter settings in the Zero Pad block are:

- **Zero pad along** = **Columns**
- **Number of output rows** = 5



Dialog Box



Zero pad along

The direction along which to pad or truncate. **Columns** specifies that the *row* dimension should be changed to M_o; **Rows** specifies that the *column* dimension should be changed to N_o; **Columns and rows** specifies that both column and row dimensions should be changed; **None** disables padding and truncation and passes the input through to the output unchanged.

Number of output rows

The desired number of rows in the output, M_o. This parameter is enabled when **Columns** or **Columns and rows** is selected in the **Zero pad along** menu.

Number of output columns

The desired number of columns in the output, N_o. This parameter is enabled when **Rows** or **Columns and rows** is selected in the **Zero pad along** menu.

See Also

Matrix Concatenation	Simulink
Pad	DSP Blockset
Repeat	DSP Blockset
Submatrix	DSP Blockset
Upsample	DSP Blockset
Variable Selector	DSP Blockset

6

DSP Function Reference

DSP Blockset Utility Functions

In addition to the blocks contained in the DSP Blockset libraries, a number of utility functions and scripts are provided in the `toolbox\dspblks\dspblks` directory. The key functions are listed below and described on the following pages:

- `dsp_links`
- `dsplib`
- `dspstartup`
- `liblinks`
- `rebuffer_delay`

Purpose

Display library link information for blocks linked to the DSP Blockset.

Syntax

```
dsp_links
dsplinks(sys)
dsplinks(sys,mode)
```

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'`.

See Also

`liblinks` DSP Blockset

dsplib

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.

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 2-12.

The `dspstartup.m` script sets the following Simulink environment parameters. See Appendix A, “Model and Block Parameters,” in the Simulink documentation for complete information about a particular setting.

Parameter	Setting
`SingleTaskRateTransMsg`	`error`
`Solver`	`fixedstepdiscrete`
`SolverMode`	`SingleTasking`

Parameter	Setting
StartTime	0.0
StopTime	inf
FixedStep	auto
SaveTime	off
SaveOutput	off
AlgebraicLoopMsg	error
InvariantConstants	on
RTWOptions	[get_param(0,'RTWOptions')', -aRollThreshold=2']

See Also `startup` MATLAB

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

rebuffer_delay

Purpose

Compute the number of samples of delay introduced by buffering and unbuffering operations.

Syntax

```
d = rebuffer_delay(f,n,m)
d = rebuffer_delay(f,n,m,'singletasking')
```

Description

`d = rebuffer_delay(f,n,m)` returns the delay (in samples) introduced by the buffering and unbuffering blocks in multitasking operations, where `f` is the input frame size, `n` is the **Buffer size** parameter setting, and `m` is the **Buffer overlap** parameter setting.

The blocks whose delay can be computed by `rebuffer_delay` are:

- Buffer
- Unbuffer

`d = rebuffer_delay(f,n,m,'singletasking')` returns the delay (in samples) introduced by these blocks in single-tasking operations.

The table below shows the appropriate `rebuffer_delay` parameter values to use in computing delay for the two blocks.

Block	Parameter Values
Buffer	`f` = input frame size (`f=1` for sample-based mode) `n` = **Buffer size** `m` = **Buffer overlap**

Unbuffer	`f` = input frame size
	`n = 1`
	`m = 0`

See Also

Buffer	DSP Blockset
Unbuffer	DSP Blockset

Index

Symbols

Numerics

A

B

C

D

E

M

N

O

P

Q

R

S

T

U

V

W

Y

Z